ModuleStudio User's Guide v0.6.2

Axel Guckelsberger and many others

August 1, 2014

Contents

I. First steps **6**

1. Introduction **7**
1.1. About ModuleStudio . . . 7
1.2. Benefits . . . 7
1.3. About this manual . . . 8
1.4. Model layers . . . 8
1.4.1. Application layer . . . 8
1.4.2. Data layer . . . 8
1.4.3. Controller layer . . . 9
1.4.4. View layer . . . 9
1.4.5. Workflow layer . . . 9
1.5. Additional notes . . . 9

2. Component overview **10**
2.1. Introduction . . . 10
2.2. Modeling Language . . . 10
2.2.1. Meta model . . . 10
2.2.2. Constraints . . . 10
2.3. Modeling Editors . . . 10
2.3.1. Graphical . . . 10
2.3.2. Textual . . . 11
2.3.3. Structural . . . 11
2.3.4. Hybrid . . . 11
2.4. Generators . . . 11
2.4.1. zClassic . . . 11
2.4.2. Reporting . . . 11
2.5. Additional notes . . . 11

3. Getting started **12**
3.1. Installation . . . 12
3.1.1. Additional notes for MacOSX . . . 12
3.2. First tour . . . 12
3.3. Development process . . . 12
3.4. Migrating old models . . . 13
3.5. Migrating old modules . . . 13

3.6. Additional notes . . . 13

4. User interface **16**
4.1. Introduction . . . 16
4.2. Basic usage . . . 16
4.3. Graphical editors . . . 16
4.3.1. Main editor . . . 16
4.3.2. Model editor . . . 16
4.3.3. Controller editor . . . 16
4.3.4. View editor . . . 16
4.3.5. Workflow editor . . . 16
4.4. Textual editors . . . 18
4.5. Useful hints . . . 18
4.5.1. Keyboard shortcuts in graphical editors . . . 18
4.5.2. Keyboard shortcuts in textual editors . . . 19
4.6. Additional notes . . . 19
4.7. Textual Grammar . . . 19

II. Enhanced topics **21**

5. Validation **22**
5.1. Introduction . . . 22
5.2. Basic usage . . . 22
5.2.1. Triggering validation . . . 22
5.2.2. Validation consequences . . . 23
5.3. Application layer . . . 23
5.3.1. Global rules . . . 23
5.3.2. Application . . . 25
5.3.3. Settings container . . . 26
5.3.4. Referred application . . . 26
5.3.5. Model container . . . 26
5.3.6. Controller container . . . 27
5.3.7. View container . . . 27
5.4. Model layer . . . 28
5.4.1. Model container . . . 28
5.4.2. Entity . . . 30
5.4.3. Entity field . . . 35
5.4.4. Relationship . . . 49
5.4.5. Entity index . . . 54
5.4.6. Variables . . . 55
5.4.7. Other elements . . . 55
5.5. Controller layer . . . 56
5.5.1. Controller container . . . 56

5.5.2. Controller . . . 57
5.5.3. Action . . . 58
5.5.4. Action handler . . . 58
5.5.5. Action event . . . 58
5.5.6. Transition . . . 59
5.6. View layer . . . 59
5.6.1. View container . . . 59
5.6.2. View . . . 59
5.6.3. Root panel . . . 60
5.6.4. Other UI elements . . . 60
5.6.5. Layout order . . . 60
5.7. Workflow layer . . . 60
5.8. Additional notes . . . 61

6. Application generator **62**
6.1. Introduction . . . 62
6.2. Basic idea . . . 62
6.3. How it works . . . 63
6.3.1. Input dialogs . . . 63
6.4. The workflow . . . 63
6.5. Generator reference . . . 64
6.5.1. Application layer . . . 65
6.5.2. Model layer . . . 72
6.5.3. Controller layer . . . 105
6.5.4. View layer . . . 109
6.5.5. Workflow layer . . . 114
6.6. Additional notes . . . 114

7. Customisation and maintenance **115**
7.1. Introduction . . . 115
7.2. Long-term maintenance . . . 115
7.2.1. Keep consistent . . . 115
7.2.2. Document your changes . . . 115
7.2.3. Use overriding . . . 115
7.2.4. Code additions . . . 116
7.2.5. Use versioning . . . 116
7.3. Additional notes . . . 116

8. Other cartridges **117**
8.1. Introduction . . . 117
8.2. Reporting . . . 117
8.2.1. Documentation . . . 117
8.2.2. Model information . . . 117
8.2.3. Function points . . . 117

8.3. Additional notes . . . 117

9. AddOns **118**
9.1. Introduction . . . 118
9.1.1. Language packs . . . 118
9.1.2. Figure galleries . . . 118
9.1.3. Template sets . . . 118
9.1.4. Generator cartridges . . . 118
9.1.5. Other extensions . . . 118
9.2. Extension Points . . . 118
9.3. Additional notes . . . 118

III. Appendix **119**

10. Troubleshooting **120**
10.1. Introduction . . . 120
10.2. General hints . . . 120
10.2.1. Pathes with space chars . . . 120
10.3. Tooling problems . . . 120
10.3.1. Boolean properties can't be unset . . . 120
10.3.2. I can't open my model again . . . 120
10.3.3. The diagram looks broken . . . 120
10.3.4. I can't save my model . . . 121
10.4. Migration problems . . . 121
10.4.1. Migrating my existing model causes an error . . . 121
10.5. Generation problems . . . 122
10.5.1. Unable to find workflow file . . . 122
10.5.2. Orphan removal with many-to-many relations . . . 122
10.5.3. The workflows are not fetched properly . . . 122
10.5.4. Patch for category selector . . . 123

11. Glossary **124**
11.1. A-C . . . 125
11.2. D-F . . . 126
11.3. G-L . . . 127
11.4. M-N . . . 129
11.5. O-R . . . 130
11.6. S-Z . . . 131

Part I.

First steps

1. Introduction

1.1. About ModuleStudio

ModuleStudio is a development environment with which one can quickly, simply and efficiently describe and generate web applications. Software developers can create complex Zikula extensions in a few steps and meet individual project requirements with them.

ModuleStudio rapidly simplifies the creation, maintenance and customisation of applications for Zikula. It speeds up this process and ensures quality of those applications at the same time. Applications can be designed or customised in a graphical editor and then the code is generated. In this way, the process is automated speeding up development time and quality many hundred fold. Maintenance and customisation also benefit from this process via graphical modeling and code generation.

Read more at this page: What is ModuleStudio

1.2. Benefits

- *Development time/costs:* avoid wasting weeks for schematical and architecturally motivated code parts!
- *Maintainability:* your software is a model - easily changeable and cheaply maintainable! No more efforts for getting your modules up to date for new versions, just regenerate and merge the code!
- *Code quality:* take profit from best practices and established patterns!
- *Architectural compliance:* take most usage from powerful core frameworks and interfaces! Generated applications follow all APIs and guidelines automatically. This is a big step for avoiding unsecure and legacy extensions when the framework itself evolves!
- *Reusability:* share and modify your models! Do not make the same work twice!
- *Understandibility:* avoid having to learn programming rules and framework details! Develop with general MVC (Model View Controller) terminology independent from technical stuff!

More information can be found in these articles:

- Advantages of ModuleStudio
- How MDSD reduces costs for long-term maintenance of comprehensive software system families

- From scaffolding and UML to MDSD and DSL

1.3. About this manual

This user manual is going to provide all required information to work with ModuleStudio. Furthermore it serves as a reference for all details of the generator. This document should be available in several formats:

- help within ModuleStudio
- online help on our website
- pdf print version

In future versions we are going to add interaction capabilities to incorporate active help into ModuleStudio itself while you are working. For now the focus was to get the manual actually written ;-)

1.4. Model layers

A ModuleStudio model is divided into several submodels in order to separate concerns. This section shows the different layers and their role for the overall application. It is just a brief overview which does not explain single fields or settings in detail. This will be done in later chapters instead.

1.4.1. Application layer

The application layer is the overall container where all sublayers are linked together. In addition it contains a bunch of application-wide properties with which general aspects are defined, like for example the name and the author of an application.

1.4.2. Data layer

Each data layer defines a logical group of data structures for the application. This involves the following key concepts:

- Entities
- Entity fields
- Join and inheritance relationships
- Behavioural and functional extensions
- Variables

1.4.3. Controller layer

Each controller layer defines a logical group of use cases for the application. This involves the following key concepts:

- Controllers
- Controller actions
- Action handlers
- Action events
- Transitions

1.4.4. View layer

Each view layer defines a logical group of view templates for the application. This involves the following key concepts:

- Views
- Root panels, composite panels, sub panels
- Forms and tables
- Activators
- Layout orders

At the moment there is no editor ready for customising the view layer. The generator will create default template sets which can be customised with overriding mechanisms offered by Zikula.

1.4.5. Workflow layer

The workflow layer is not implemented yet. This section is just a dummy for future.

1.5. Additional notes

None yet.

2. Component overview

2.1. Introduction

This page gives you an overview of the main parts of ModuleStudio and how they work together.

2.2. Modeling Language

The inner core of MOST is a domain-specific language (DSL) for Zikula extensions. This language allows a formalised description of applications which is a fundamental requirement to process models automatically with the help of transformations.

For convenience ModuleStudio uses general terms for modeling MVC applications. As you will see later an application model has different submodels for describing corresponding architectural layers (model, controller, view).

2.2.1. Meta model

The meta model defines the essential concepts of the ModuleStudio language, that is which model elements may exist and how they are allowed to work with each other. This allows reusing the basic domain concepts at several places, like validation, editors, generators and so on.

2.2.2. Constraints

In addition to the meta model there are many validation rules (§5) to enrich the modeling language with more precise knowledge. These constraints ensure that the generator can only be started for valid models.

2.3. Modeling Editors

The user interface consists of different types of editors which may include event different kinds of how information is described.

2.3.1. Graphical

Graphical notations are convenient for modeling edges between different nodes. They are not that well suited for creating huge lists of similar elements for instance.

ModuleStudio offers graphical editors for creating and changing models for describing different applications. See the user interface chapter (§4) for more information.

2.3.2. Textual

A textual syntax is very nice for rapid creation of structures. It becomes less handy for relationships.

At the moment there is no textual editor included in ModuleStudio. This is going to change soon though.

2.3.3. Structural

Structural views, for example trees, are another possible viewpoint for describing a model.

At the moment there is no structural editor included in ModuleStudio. This may change in future though.

2.3.4. Hybrid

In future of ModuleStudio is heading towards some kind of hybrid modeling where you can combine textual and graphical editors for describing applications.

2.4. Generators

The generators add technical details depending on the target system. Their task is creating source code or other artifacts from application models.

2.4.1. zClassic

The primary generator cartridge is *zclassic* which creates a Zikula extension. You can read more about this in the generator chapter (§6). Also important is another chapter about customisation and maintenance (§7) of generated applications.

2.4.2. Reporting

The *reporting* cartridge creates some documents from a given model. Please continue here for details (§8.2).

2.5. Additional notes

None yet.

3. Getting started

This chapter explains the first steps required for starting creating applications with ModuleStudio.

For now we refer only to existing tutorials as they describe things still quite well.

3.1. Installation

Simply download the archive for your operating system and extract it inside your home directory. After that you are ready to start the ModuleStudio executable.

Note ModuleStudio requires Java in at least version 7. If no JRE (Java Runtime Environment) is found on your system path, a dialog appears and you are suggested to download and install the JRE before starting ModuleStudio.

3.1.1. Additional notes for MacOSX

If ModuleStudio does not start but fails with an error please check the created error log file. In case it contains lines like *!MESSAGE Missing required capability RequireCapability: osgi.ee; filter="(&(osgi.ee=JavaSE)(version=1.7))".* then the system is not using Java (Standard Edition) 7. To verify this open a console and type: *java -version.* If Java 7 is used in your system by default the output should show something starting with 1.7.x.

According to feedback from our users it is also important to install the MacOS JDK instead of the JRE, since the JDK installer properly tells OSX to use 1.7 where the JRE does not.

3.2. First tour

First thing you may want to do after starting the ModuleStudio application is creating a new model. Therefore start the *new application wizard* using the "File" main menu. It allows you to enter some basic information, like vendor and application name, in a simple form. After that it creates a new model and preinserts some common basic container elements so that you can directly proceed with describing your application in detail.

Please see this tutorial for getting the overall idea.

3.3. Development process

Developing model-driven applications works well in combination with an iterative-incremental development process. In this approach you start with a small model which will then be

enhanced in several steps whereby some short tests verify that the direction is correct.

Each cycle consists of the following steps:

1. Create or change model
2. Regenerate
3. Merge changes
4. Test intermediate results

3.4. Migrating old models

To open a model from an older version please remove all account controllers and search controllers first.

Afterwards the existing model can be converted by using the menu entry "File > Migrate old model from 0.5.x".

3.5. Migrating old modules

If you want create a model for an existing legacy module you can follow the following procedure.

1. First get the file *pntablesToXML.php* from GitHub.
2. Use this script to migrate your old *pntables.php* file to an xml file.
3. Inside ModuleStudio use the menu entry "File > Import xml table definition" which will open a file selection dialog. Choose the xml file you just created. As a result you will get a new application model in the *MOST_output/yourmod.mostapp* file.
4. Now use "File > Initialize diagram file" to open this application model and create a new diagram file for it.
5. In the data editor remove unrequired elements, like table prefixes and default identifier fields, that are primary and foreign keys.
6. Follow validation messages to get remaining stuff sorted properly.

Example import results:

3.6. Additional notes

None yet.

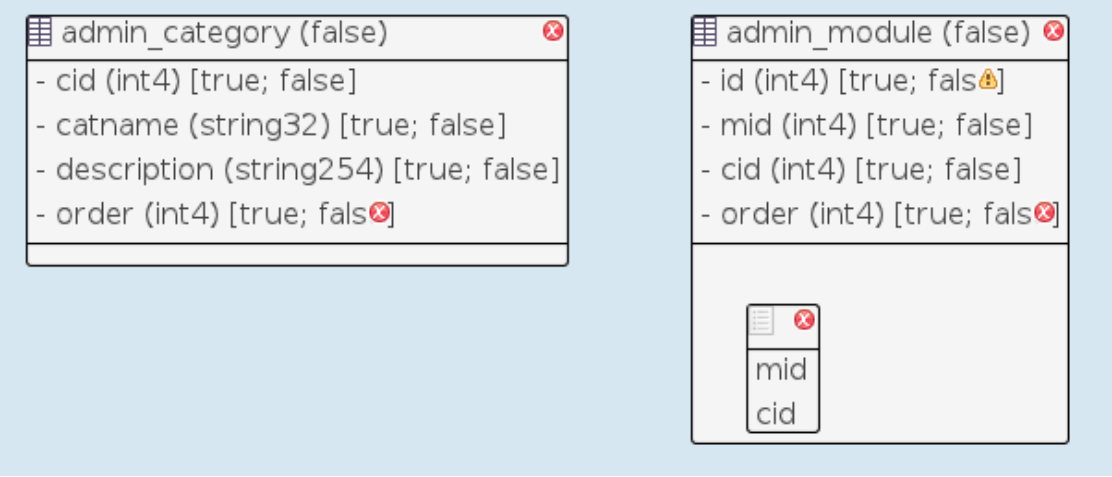

Figure 3.1.: Admin module

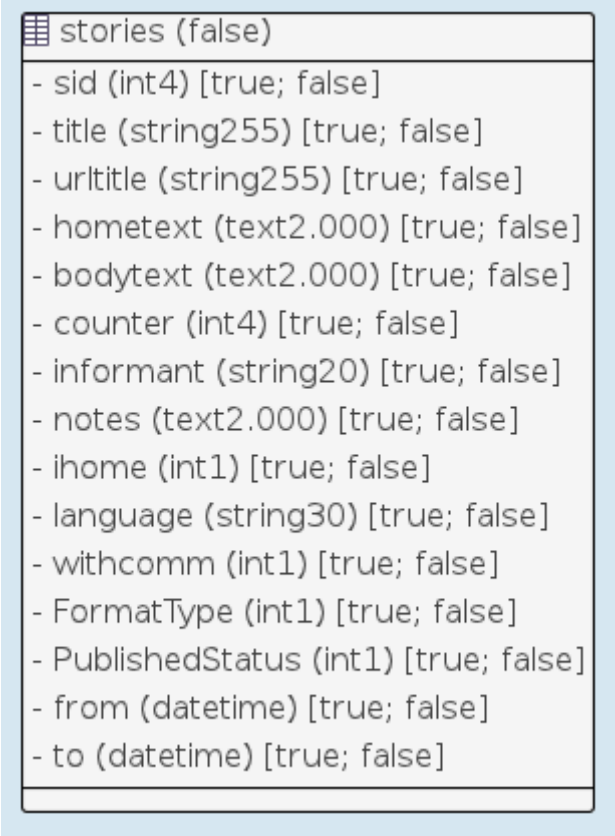
stories (false)
- sid (int4) [true; false]
- title (string255) [true; false]
- urltitle (string255) [true; false]
- hometext (text2.000) [true; false]
- bodytext (text2.000) [true; false]
- counter (int4) [true; false]
- informant (string20) [true; false]
- notes (text2.000) [true; false]
- ihome (int1) [true; false]
- language (string30) [true; false]
- withcomm (int1) [true; false]
- FormatType (int1) [true; false]
- PublishedStatus (int1) [true; false]
- from (datetime) [true; false]
- to (datetime) [true; false]

Figure 3.2.: News module

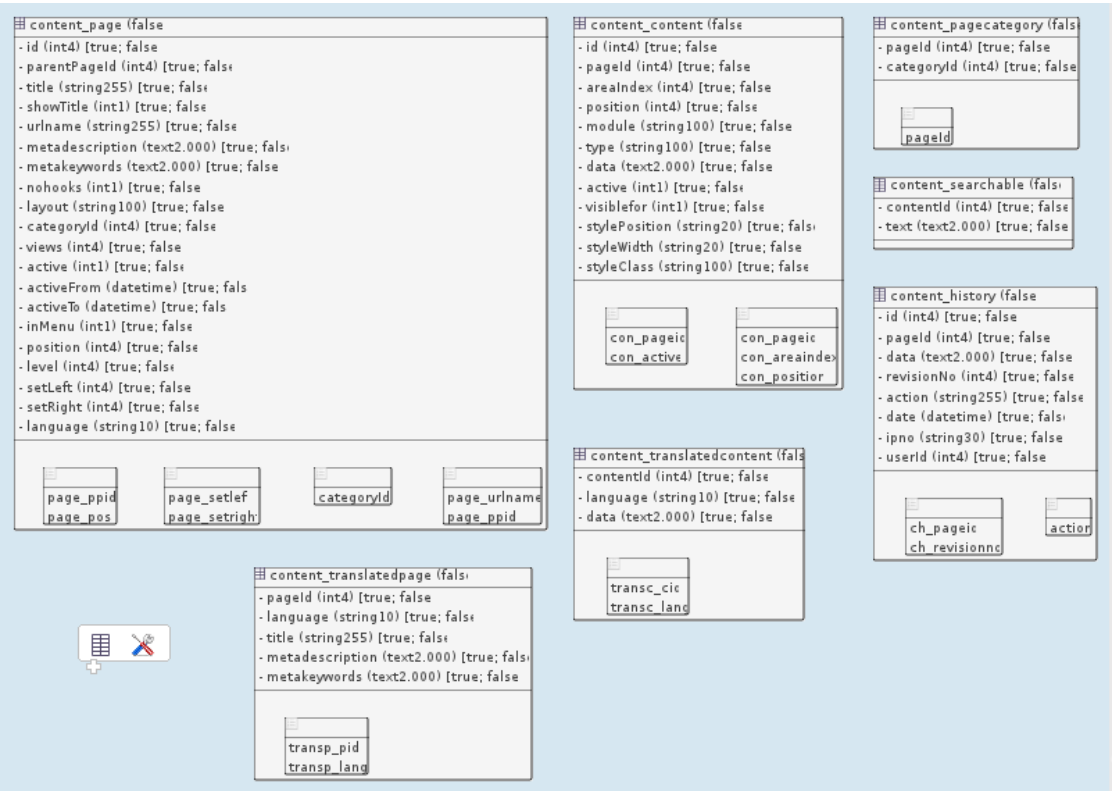

Figure 3.3.: Content module

4. User interface

4.1. Introduction

This section shows how to use ModuleStudio. Starting with a general demonstration of the user interface it goes step by step through all single editors required for creating a complete model.

At the moment this page consists only of a few links to corresponding tutorials. It will be enhanced at a later stage after the actual modeling language has matured enough to spend work on completing the user interface.

4.2. Basic usage

Please see this tutorial.

4.3. Graphical editors

4.3.1. Main editor

Please see this tutorial.

4.3.2. Model editor

Please see this tutorial.

4.3.3. Controller editor

Please see this tutorial.

4.3.4. View editor

The view editor has not been implemented yet. This section is therefore just a dummy for future.

4.3.5. Workflow editor

The workflow editor has not been implemented yet. This section is therefore just a dummy for future.

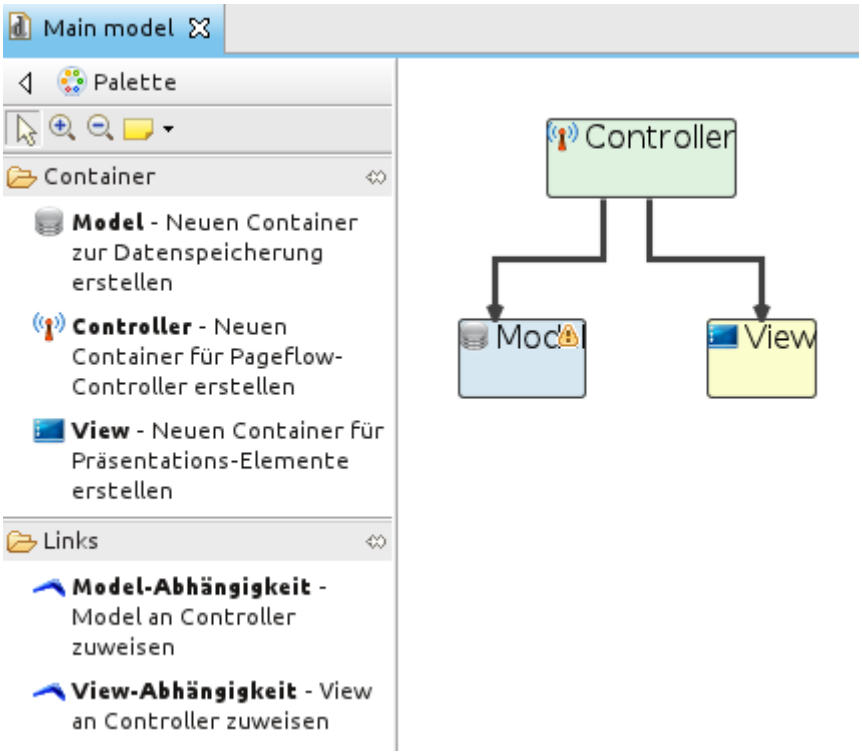

Figure 4.1.: Main model

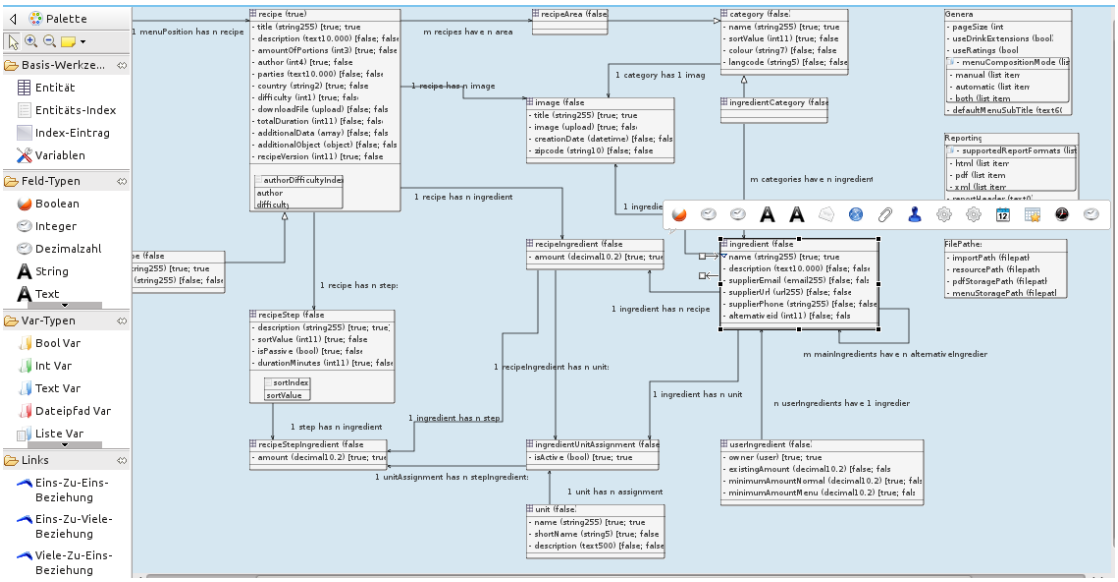

Figure 4.2.: Model model

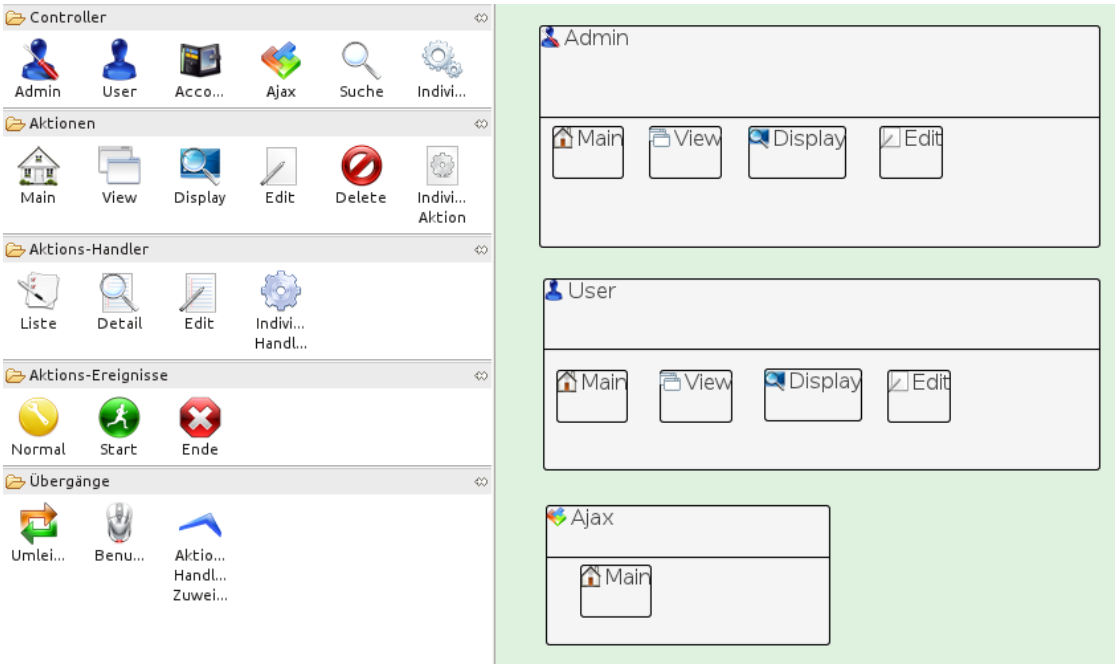

Figure 4.3.: Controller model

4.4. Textual editors

Beginning with ModuleStudio 0.6.0 there is also a textual syntax notation available. Not visible at once, it will be integrated into the UI step by step.

4.5. Useful hints

Here are some tutorial showing special abilities for certain use cases:

- Customise palette
- Multiple container elements
- Moving fields with drag n drop
- Creating multiple elements quickly
- Working with multiple windows

4.5.1. Keyboard shortcuts in graphical editors

There are some very handy shortcuts hidden in ModuleStudio. For example it can be worth to experiment with the Ctrl (Alt on Mac) and/or Shift keys when moving or resizing an object.

Here is a list of all editor shortcuts.

If one has selected a palette tool and creates an object in the diagram it is usually required to select the tool again in order to create another object. If one presses the Ctrl (Alt on Mac) key one can create multiple elements of the same type in one step.

4.5.2. Keyboard shortcuts in textual editors

The following list shows some basic commands which might be helpful when using the textual editors.

Shortcut	Action description
Alt-Up / Alt-Down	Move current line or selection one line up / down.
Alt-Left / Alt-Right	Go back / forward in the history of editors.
Alt-Shift-Up / Alt-Shift-Down	Expand selection to containing element.
Alt-Shift-R	Rename current element as well as all other occurences.
F3 or Ctrl-MouseClick	Follow reference under cursor.
Ctrl-Up / Ctrl-Down	Scroll one line up / down.
Ctrl-PgUp / Ctrl-PgDown	Activate previous / next editor tab.
Ctrl-0	Pop up outline for easy navigation and filtering.
Ctrl-1	Quick fix of errors.
Ctrl-/	Toggle comment for line or selection.
Ctrl-Space	Start content assist suggesting possible values.
Ctrl-D	Delete current line.
Ctrl-L	Go to a certain line.
Ctrl-M	Maximize current editor window.
Ctrl-W	Close current editor tab.
Ctrl-Q	Go to last edit location.
Ctrl-Shift-F	Start the source code formatter.
Ctrl-Shift-G	Find references to current element.
Ctrl-Shift-F3	Locate a certain element.

4.6. Additional notes

None yet.

4.7. Textual Grammar

Here is a railroad chart showing the textual grammar elements:

Figure 4.4.: Textual grammar

Part II.

Enhanced topics

5. Validation

This chapter explains the intention behind validation in ModuleStudio. The biggest part is a reference section listing all validation rules in detail and explains the motivation behind them. You can use this reference to search for certain error messages if you want to know more about the background. If you are unsure about the terminology used in a description please refer to the generator chapter (§6) in this documentation.

5.1. Introduction

Validation is an essential part of a modeling language. By checking the current model against several *constraints* it ensures that certain scenarios can not happen in order to avoid problems based on invalid states of model elements.

With the help of validation in ModuleStudio all subsequent components can process the given application model without having to revalidate common concerns.

Having clean and properly validated models is also very helpful for Zikula as a framework because in the long term this leads to a constantly high quality across third party extensions which is traditionally a weak point for every open source system. See this tutorial for more information about this aspect.

The constraints described here are evaluated within the graphical editor as well as during headless generator workflows.

5.2. Basic usage

This section gives a brief overview of how validation can be executed and what can be done if problems occur.

5.2.1. Triggering validation

As validation shows what remains to be done in order to generate an application out of a given model, it is essential to provide convenient means for checking validation constraints.

ModuleStudio validates a model every time you save it in the main editor. Validation is also executed after an existing model has been opened.

In addition you can always start the validator manually using the main menu which might be advantageous to get immediate feedback after having done some amendments in the sub editors.

Earlier versions of ModuleStudio used to perform a scheduled validation. This has been removed to save performance.

5.2.2. Validation consequences

As the generator requires a model without validation problems it can only be started in the main menu after your model has no errors left.

If the menu entry for starting the generator is inactive even if your models seems to be clean, please save the model in the main editor and click once in the diagram canvas to let the validation update it's state accordingly.

In case live validation has been deactivated you can start a *manual validation* in the main menu, too.

5.3. Application layer

5.3.1. Global rules

Context	Check	Remarks
Every element with a name	The name must be a valid identifier (e.g. no whitespace or special characters).	Camel case is recommended to get more readable messages generated, for example *mySpecialUser*

5.3.2. Application

Context	Check	Remarks
Application	The application must have a name.	Application name must have a length of at least five chars.
Application	Application name must start with a capital letter.	Since Zikula 1.3 extensions must start with a capital letter. Of course we could simply generate it this way. But for consistency we decided to follow this rule more strictly by enforcing it in the model already.
Application	The application must have an author.	
Application	The application must have an email address.	
Application with email set	The value for the application email field must be a valid email address.	
Application with url set	The value for the application url field must be a valid url.	Protocols allowed per default are http and https.
Application	The application must have a prefix for it's database tables.	This prefix is required to prevent naming collisions between several modules. Otherwise it would be a problem if multiple extensions use common table names like *user* or *category*. The prefix must have a length of more than two chars, whereby a at least four is recommended.
Application	The prefix must be a valid identifier (e.g. no whitespace or special characters).	Essentially the same as the global rule for names above. You should use lowercase here, but it will be generated in lowercase in all cases.
Application	The application must have a version. The application version must conform to the pattern x.y.z.	Another requirement introduced in Zikula 1.3. Valid values are 1.0.0, 1.2.2, but not 1.1 or 2.1.0beta.
Application	The application must contain at least one model container.	At the moment ModuleStudio wants a model with at least one entity. If you are modeling an extension without any data storage, just create some dummy elements.
Application with model containers	Names of model containers must be unique.	Unique identification and separation of data sources.
Application with model containers	There should be at least one model container refering to the system's internal data source. There must be only one model container refering to the system's internal data source.	Each model container has a property marking it as default data source or not. Only one container can set this to true which will then be handled as normal by Zikula. Additional model containers

5.3.3. Settings container

Context	Check	Remarks
Settings container	Settings container must be assigned to an application.	Should not occur in practice, this is just for completeness.
Settings container	Scribite plugins require external controller and finder component.	

5.3.4. Referred application

Context	Check	Remarks
Referred application	The imported application must have a name.	Use the real name of the module being imported.
Referred application	Please specify the minimum version for the dependency. The minimum version must conform to the pattern x.y.z.	
Referred application	Please specify the maximum version for the dependency. The maximum version must conform to the pattern x.y.z.	
Referred application	The minimum version must not be greater than the maximum version.	
Referred application	You need to specify the URI of the imported application's model file.	

5.3.5. Model container

Context	Check	Remarks
Model container	The model container must have a name.	The container name must have a length of at least three chars. Recommended are at least four chars.
Model container	Model container must be assigned to an application.	Should not occur in practice, this is just for completeness.
Application with model container with entities	Exactly one entity must be declared as leading (leading=true).	More information in the Entity section (§5.4.2).

5.3.6. Controller container

Context	Check	Remarks
Controller container	The controller container must have a name.	The container name must have a length of at least three chars. Recommended are at least four chars.
Controller container	Controller container must be assigned to an application.	Should not occur in practice, this is just for completeness.
Controller container	Every controller container must rely on at least one model container.	You have to create relationships between controller and model containers. This specifices which controller functions may process data from which sources.
Controller container	Every controller container must reference at least one view container.	You have to create relationships between controller and view containers. This specifices which controller functions may use which view templates for their output. This rule is currently not active as the view editor is not ready yet.
Application with controller container with controllers	There must not exist more than one (admin | user | ajax) controller.	Predefined controllers are unique per application (i.e. there is only one admin area).
Application with controller container with controllers	Names of custom controllers must be unique.	For example there may not be two controllers which are both named *edit*.

5.3.7. View container

Context	Check	Remarks
View container	The view container must have a name.	The view name must have a length of at least three chars. Recommended are at least four chars.
View container	View container must be assigned to an application.	Should not occur in practice, this is just for completeness.

5.4. Model layer

5.4.1. Model container

Context	Check	Remarks
Model container	The model container must contain at least one entity.	
Model container with entities	Entity names must be unique.	For example there may not be two entities which are both named *person*.
Model container with entities	The amount of entities is getting quite high. Maybe it makes sense to split up the model into two single applications.	This warning appears if you have more than 14 entities. Remember Zikula is a modular system. You can design whole families of extensions with ModuleStudio, so please try keeping complexity low and apply separation of concerns.
Application with model container with entities	An entity must not have the same name as the application.	This case is reserved as it could make sense to use corresponding namespaces in generation for encapsulating some common code parts.
Model container	It is recommended to add two integer variables named *pageSize* and *showOnlyOwnEntries* for controlling the view action's behaviour.	Occurs if a model's controllers contain a view action, but no according variables are found in the (any) model layer.

5.4.2. Entity

General entity settings

Context	Check	Remarks
Entity	The entity must be assigned to a model container.	Should not occur in practice, this is just for completeness.
Entity	Every entity must have a (name | name for multiple instances).	Entity (multiple) name must have a length of at least two chars. Should have at least four chars.
Entity	Entity (multiple) name must not contain underscores.	Underscores are not allowed as they are used for class autoloading.
Entity	Every table must contain at least one field or inherit fields from a parent table.	Ensures that either there are some fields in this entity or an outgoing inheritance relationship.
Entity with fields	You may not mark a field as primary, this is done automatically - unless maybe you want to create composite primary keys.	This warning appears because ModuleStudio adds primary keys automatically and uses the Doctrine 2 default settings if nothing else is explitely specified in the model. Beside special use cases like custom join conditions or composite primary keys you won't need to set primary keys manually.
Entity with fields	Remove ID fields... you do not need them ;-) unless maybe you want to create composite primary keys.	This warning appears if a field is named like *id* or *personid* or *person_id* for an entity named *person*. In these cases ModuleStudio adds primary keys to this table automatically before the generation.
Entity with composite primary keys	Composite entities can only have identifier strategy NONE.	Automatic identifier generation is not possible for composite primary keys.
Entity with fields	The entity needs a display pattern.	Occurs when no display pattern is defined.
Entity with fields	The display pattern does not contain any field references. Please check whether this is on purpose or not.	This warning occurs if a display pattern does not contain any field references.
Entity with archive	Every entity with an archive (*hasArchive* set to *true*) must contain one datetime or date field designated as end date.	For implementation of an automatic archiving functionality the system must know when an object has reached the end of it's lifecycle.

Inheritance-related entity settings

Context	Check	Remarks
Entity with (inheritance) relationships	All entities within a class hierarchy must have the same change tracking policy.	Requirement by Doctrine 2.
Entity with (inheritance) relationships	All entities within a class hierarchy must not have a field with the same name.	
Entity	All inheritance connections within a class hierarchy must have the same inheritance strategy.	
Entity	All inheritance connections within a class hierarchy must have the same discriminator column.	
Entity	An entity can not inherit from multiple entities.	
Entity defined as mapped super class	All associations outgoing from mapped super classes must be unidirectional.	
Entity defined as mapped super class	One-to-many relations are not possible for mapped super classes.	
Entity defined as mapped super class	Many-to-many relations are only possible for mapped super classes if it is used only in one entity at the same time.	
Entity	The length of all entity fields must not be higher than 21845.	The limit is 65535 bytes, while UTF-8 requires three bytes for each char.
Entity	Entities with owner permissions need standard fields activated.	The standard fields extension is required to determine the owner (createdUser) of an object.

Extension-related entity settings

Context	Check	Remarks
Entity defined as loggable	Loggable entities need one field with the version attribute set to true.	Can be either integer or datetime fields.
Entity defined as geographical	Entities with geographical behaviour should ideally contain a String Field with name zipcode with a length of at least 10.	Just a warning to support best practices.
Entity defined as loggable	There must not exist an entity named FooLogEntry as this is reserved by the corresponding extension.	For an entity named *person* ModuleStudio does not only generate corresponding *Person* classes, but also an entity named *PersonLogEntry* for managing the version log entry entities.
Entity with translatable field	There must not exist an entity named FooTranslation as this is reserved by the corresponding extension.	For an entity named *person* ModuleStudio does not only generate corresponding *Person* classes, but also an entity named *PersonTranslatable* for managing translation entities.
Entity defined with closure tree	There must not exist an entity named FooClosure as this is reserved by the corresponding extension.	For an entity named *person* ModuleStudio does not only generate corresponding *Person* classes, but also an entity named *PersonClosure* for managing the closure entities.
Entity defined as attributable	There must not exist an entity named FooAttribute as this is reserved by the corresponding extension.	For an entity named *person* ModuleStudio does not only generate corresponding *Person* classes, but also an entity named *PersonAttribute* for managing the attribute entities.
Entity defined as categorisable	There must not exist an entity named FooCategory as this is reserved by the corresponding extension.	For an entity named *person* ModuleStudio does not only generate corresponding *Person* classes, but also an entity named *PersonCategory* for managing the category entities.
Entity with meta data	There must not exist an entity named FooMetaData as this is reserved by the corresponding extension.	For an entity named *person* ModuleStudio does not only generate corresponding *Person* classes, but also an entity named *PersonMetaData* for managing the meta data entities.

5.4.3. Entity field

General field settings

Context	Check	Remarks
Entity field	Field must be assigned to an entity.	Should not occur in practice, this is just for completeness.
Entity field	Every field must have a name.	Field name must have a length of at least two chars. Should have more than three chars.
Entity field	Field names must be unique.	
Entity field	Field name is a reserved identifier (module, type, func, lang, theme).	These are reserved vars in traditional Zikula extensions.
Entity field	Field name is a reserved identifier (_controller, _method, _locale).	These are reserved vars in the Symfony framework.
Entity field	Field name is a reserved identifier (workflowState).	This list field is added automatically by a model-to-model transformation before the actual generation happens.
Entity field	Field name is a reserved database keyword.	ModuleStudio prevents the usage of keywords which are reserved in some database systems. Background is that there are no column prefixes anymore. For a list of all keywords see the following section (§5.4.3).
Entity field	Mandatory fields may not be nullable, too.	Occurs if you try to activate both *mandatory* and *nullable* properties for a field.
Entity field	The default value is too long for the specified field length.	
Entity field	The minimum value is too long for the specified field length.	
Entity field	The maximum value is too long for the specified field length.	
Entity field	Fields using text or blob data types must not be unique (applies for text, array and object fields).	

Note all rules for name validation are applied not only to name, but also to the dbName property, too, if it is set for a certain field.

Reserved database keywords

The following list has been merged and includes therefore all keywords of all supported DBMS.

- abort, access, action, activate, add, after, alias, all, allocate, allow, alter, analyse, analyze, and, any, arraylen, as, asc, asensitive, associate, asutime, at, attach, attributes, audit, authorization, autoincrement, aux, auxiliary, avg
- backup, before, begin, between, bigint, binary, blob, both, break, browse, bufferpool, bulk, by
- cache, call, called, capture, cardinality, cascade, cascaded, case, cast, ccsid, change, char, character, check, checkpoint, clone, close, cluster, clustered, coalesce, collate, collection, collid, column, comment, commit, committed, compress, compute, concat, condition, conflict, connect, connection, confirm, constraint, contains, containstable, continue, controlrow, convert, count, count_big, create, cross, current, current_date, current_lc_ctype, current_path, current_schema, current_server, current_time, current_timestamp, current_timezone, current_user, cursor, cycle
- data, database, databases, datapartitionname, datapartitionnum, date, day, days, day_hour, day_microsecond, day_minute, day_second, db2general, db2genrl, db2sql, dbcc, dbinfo, dbpartitionname, dbpartitionnum, deallocate, dec, decimal, declare, default, defaults, deferrable, deferred, definition, delayed, delete, dense_rank, denserank, deny, desc, describe, descriptor, detach, deterministic, diagnostics, disable, disallow, disconnect, disk, distinct, distinctrow, div, distributed, do, document, double, drop, dssize, dummy, dual, dynamic
- each, editproc, else, elseif, enable, encoding, encryption, end, enclosed, end-exec, ending, erase, errlvl, errorexit, escape, escaped, every, except, exception, excluding, exclusive, exec, execute, exists, exit, explain, external, extract
- fail, false, fenced, fetch, fieldproc, file, final, fillfactor, float, float4, float8, floppy, for, force, foreign, free, freetext, freetexttable, freeze, from, full, fulltext, function
- general, generated, get, glob, global, go, goto, grant, graphic, group
- having, handler, hash, hashed_value, having, high_priority, hint, hold, holdlock, hour, hours, hour_microsecond, hour_minute, hour_second
- identified, identity, identitycol, identity_insert, if, ignore, ilike, immediate, in, including, inclusive, increment, index, indexed, indicator, inf, infile, infinity, inherit, initial, initially, inner, inout, insensitive, insert, instead, int, int1, int2, int3, int4, int8, integer, integrity, intersect, interval, into, is, isnull, isobid, isolation, iterate
- jar, java, join

- keep, key, keys, kill
- label, language, lateral, lc_ctype, leading, leave, left, level, like, limit, lineno, lines, linktype, load, local, localdate, locale, localtime, localtimestamp, locator, locators, lock, lockmax, locksize, long, longblob, longtext, loop, low_priority
- maintained, match, materialized, max, maxextents, maxvalue, mediumblob, mediumint, mediumtext, microsecond, microseconds, middleint, min, minus, minute, minutes, minute_microsecond, minute_second, minvalue, mirrorexit, mod, mode, modifies, modify, month, months
- nan, national, natural, new, new_table, nextval, no, noaudit, nocache, nocheck, nocompress, nocycle, nodename, nodenumber, nomaxvalue, nominvalue, nonclustered, none, noorder, normalized, not, notfound, notnull, nowait, no_write_to_binlog, null, nullif, nulls, number, numeric, numparts
- obid, of, off, offline, offset, offsets, old, old_table, on, once, online, only, open, opendatasource, openquery, openrowset, optimization, optimize, option, optionally, or, order, out, outer, outfile, over, overlaps, overriding
- package, padded, pagesize, parameter, part, partition, partitioned, partitioning, partitions, password, path, pctfree, percent, perm, permanent, piecesize, placing, pipe, plan, pragma, position, precision, prepare, prevval, primary, print, prior, priqty, privileges, proc, procedure, processexit, program, psid, public, purge
- query, queryno
- raid0, raise, raiserror, range, rank, raw, read, reads, readtext, real, reconfigure, recovery, references, referencing, refresh, regexp, reindex, release, rename, repeat, repeatable, replace, replication, require, reset, resignal, resource, restart, restore, restrict, result, result_set_locator, return, returns, revoke, right, rlike, role, rollback, round_ceiling, round_down, round_floor, round_half_down, round_half_even, round_half_up, round_up, routine, row, rowcount, rowguidcol, rowid, rowlabel, rownum, rownumber, rows, rowset, row_number, rrn, rule, run
- save, savepoint, schema, schemas, scratchpad, scroll, search, second, seconds, second_microsecond, secqty, security, select, sensitive, separator, sequence, serializable, session, session_user, set, setuser, share, show, shutdown, signal, similar, simple, size, smallint, snan, some, soname, source, spatial, specific, sql, sqlbuf, sqlexception, sqlid, sqlstate, sqlwarning, sql_big_result, sql_calc_found_rows, sql_small_result, ssl, stacked, standard, start, starting, statement, static, statistics, statment, stay, stogroup, stores, straight_join, style, substring, successful, sum, summary, synonym, sysdate, sysfun, sysibm, sysproc, system, system_user
- table, tablespace, tape, temp, temporary, terminated, textsize, then, time, timestamp, tinyblob, tinyint, tinytext, to, top, trailing, tran, transaction, trigger, trim, true, truncate, tsequal, type
- uid, uncommitted, undo, union, unique, unlock, unsigned, until, update, updatetext, usage, use, user, using, utc_date, utc_time, utc_timestamp

- vacuum, validate, validproc, value, values, varbinary, varchar, varchar2, varcharacter, variable, variant, varying, vcat, verbose, version, view, virtual, volatile, volumes
- waitfor, when, whenever, where, while, with, without, write, writetext, work
- x509, xmlelement, xmlexists, xmlnamespaces, xor
- year, years, year_month
- zerofill

Extension-related field settings

Context	Check	Remarks
Entity field	Entities with sluggable behaviour may not include a field named slug.	If a field is named *slug* it gets this error as soon as at least one field in this entity has set the *sluggable position* attribute to a number greater than 0. ModuleStudio creates this slug field automatically in addition to the other fields in the model.
Entity field	The sluggable position values must be unique per entity.	If fields are included into a slug by setting a value greater than 0, this value must be unique per entity. The position defines in which order the field values are considered as permalink parts.
Entity field	Only one field per entity may store the sortable position.	Can occur if one tries to use multiple integer or user fields as position for the sortable extension.
Entity field	The sortable position may not be the sortable group, too.	As soon as a field is used as sortable position it can not also act as the grouping criteria at the same time.
Entity field	You need another field as sortable position to make the sortable group work.	Can occur for a field marked as sortable group.
Entity field	Only one field per entity may store the version.	Can appear for integer and datetime fields if you enabled the *version* property for more than one field in the same entity.
Entity field	Only one field per entity may represent a start date.	Can appear for datetime and date fields if you enabled the *startDate* property for more than one field in the same entity.
Entity field	Only one field per entity may represent an end date.	Can appear for datetime and date fields if you enabled the *endDate* property for more than one field in the same entity.
Entity field	Entities with a version field should use optimistic locking.	Can appear for integer and datetime fields with the *version* property enabled. You must set the locking type of the corresponding entity to either *OPTIMISTIC* or *PAGELOCK_OPTIMISTIC*, depending on whether you want support the Zikula PageLock functionality in addition or not.

Numeric fields

Context	Check	Remarks
Integer and user fields	The default value for an integer field must contain only digits.	
Integer field	The maximum length for an integer field is 18.	Corresponds to a *bigint* mapping in Doctrine 2.
Integer field	Minimum value must not be larger than maximum value.	
Integer field	Entities with an aggregate field should use a locking strategy (optimistic or pessimistic read).	If an integer field acts as aggregate field the corresponding entity must use one locking strategy of *OPTIMISTIC*, *PAGELOCK_OPTIMISTIC*, *PESSIMISTIC_READ*, *PAGELOCK_PESSIMISTIC_READ*, depending on whether you want support the Zikula PageLock functionality in addition or not.
Integer field	Aggregate fields work only in combination with an outgoing and bidirectional one-to-many relationship with persist cascade.	
Integer field	Naming of aggregateFor attribute values must follow the syntax targetAlias#targetFieldName (for example views#amount).	If an integer field acts as aggregate field the property *aggregate for* must define the target alias of corresponding outgoing and bidirectional one-to-many relationship with persist cascade. After a *# char* as delimiter the name of the target field (to be aggregated) follows.
Decimal field	The default value for a decimal field must be a floating point number.	
Decimal field	The length (precision) of a decimal field must be greater than the scale.	For example *1234.12* is valid ($4 > 2$), but *123.1234* is not.
Decimal field	Minimum value must not be larger than maximum value.	
Float field	The default value for a float field must be a floating point number.	
Float field	Minimum value must not be larger than maximum value.	

String and text fields

Context	Check	Remarks
All string fields	String size must not be smaller than 1.	Occurs if length is set to 0.
All string fields	String size must be larger than minimum length.	If you set a minimum length it must not be larger than the actual field length.
String field	String length for credit card numbers must be at least 20 chars.	Occurs if you activate the *creditCard* property for a field with a length smaller than 20.
String field	String length for country codes must be at least 2 chars.	Occurs if you activate the *country* property for a field with a length smaller than 2.
String field	String length for currency codes must be at least 3 chars.	Occurs if you activate the *currency* property for a field with a length smaller than 3.
String field	String length for languages must be at least 7 chars.	Occurs if you activate the *language* property for a field with a length smaller than 7.
String field	String length for locales must be at least 5 chars.	Occurs if you activate the *locale* property for a field with a length smaller than 5.
String field	String length for HTML colour codes must be at least 7 chars.	Occurs if you activate the *html colour* property for a field with a length smaller than 7.
String field	String length for ip addresses must be at least 15 chars.	Occurs if you set the *ipAddress* property to a value covering IPv4 addresses for a field with a length smaller than 15.
String field	String length for ip addresses must be at least 39 chars.	Occurs if you set the *ipAddress* property to a value covering IPv6 addresses for a field with a length smaller than 39.
String field	String length for uuids must be at least 36 chars.	Occurs if you activate the *uuid* property for a field with a length smaller than 36.
String field	A string can only be one of country, creditCard, currency, language, locale, htmlcolour, iban, isbn, issn, ip address, password and uuid.	
String field	String fields for countries, currencies, languages, locales, ip addresses, colours and uuids must also activate the nospace validator.	The *nospace* property ensures that spaces are not allowed as part of the input value. The generator could use it without having set this to true, but as the setting is there anyway the proper solution is to enforce the user activating it for consistency.
String field	String length must not be	

Date and time fields

This section includes rules which apply only for datetime, date and time fields.

Context	Check	Remarks
All date fields	A value can not be in the past and in the future at the same time.	You can only activate either *past* or *future* validators for one field.
Datetime and date fields	A value can not represent both start and end date at the same time.	You can only activate either *startDate* or *endDate* flags for one field.
All date fields	The timestampable change trigger field must point to workflowState, the name of a field or a relation (property.field).	If the *timestampable* property for a field has been set to *CHANGE* then this error can appear to remind you to set also the required attribute *timestampableChangeTriggerField*. Either you did not set anything there or the value does neither correspond to workflowState nor the name of an entity field nor an incoming relation.
Datetime field	It would be preferable to use an integer column as version field, as datetimes could potentially lead to conflicts for high traffic sites depending on the timestamp resulution in the database.	If you use datetime fields with *version* set to true this warning will appear. As the Doctrine 2 manual points out integers are more robust against race conditions in high traffic environments where timestamp comparisons are limited due to how precise the used database does it. Therefore you should prefer integer fields for storing versions except side conditions require the usage of datetime fields for that.
Datetime field	The default value for a datetime field must conform to the pattern YYYY-MM-DD HH:MM:SS or now.	You can either set a certain value or use *now* to specify the current timestamp.
Date field	The default value for a date field must conform to the pattern YYYY-MM-DD or now.	You can either set a certain value or use *now* to specify today.
Time field	The default value for a time field must conform to the pattern HH:MM:SS or now.	You can either set a certain value or use *now* to specify the current moment.

Other fields

Upload field	The allowedExtensions attribute must contain a comma separated list of the file types to be allowed during the upload (example: gif, jpeg, jpg, png). Note that the separator is ', ' including the space char.	
Upload field	There must not exist a field named fooMeta because this is reserved for an automatic field storing meta data for this upload.	
List field	The list must contain at least one item.	
List field	The list must contain at least two items.	Occurs if you enable *multiple* without having defined multiple list items.
List field	The list is not set to multiple and may therefore only contain one item with default property set to true.	
List field	Only lists with multiple selections can define an enforced minimum (maximum) amount of selected values.	Occurs if you set *min* (*max*) to a value not equal to *0*, without having set *multiple* to *true*.
List field	The amount of list items must not be smaller than the minimum (maximum) amount of enforced values.	Occurs if you set *min* (*max*) to a value which is greater than the actual number of existing list items.
List field	The maximum amount of enforced values must not be smaller than the minimum amount of enforced values.	Occurs if you set *max* to a smaller value than *min*.
List field	Only lists with multiple selections can consist of checkboxes.	Occurs if you set *useChecks* to *true*, but not *multiple*. Checkbox lists with only one selection allowed are postponed to a future version, since you can use a boolean field for this, too.
Array field	Enforcing a certain amount of items requires setting both *min* and *max* properties.	Occurs if you set either *min* or *max* to a value greater than *0*, but not both.
Array field	The maximum amount of enforced items must not be smaller than the minimum amount of enforced items.	Occurs if you set *max* to a smaller value than *min*.

5.4.4. Relationship

Context	Check	Remarks
Relationship	Relation must be assigned to a model container.	Should not occur in practice, this is just for completeness.
Relationship	Every relation must have a (source | target) entity.	Should not occur in practice, this is just for completeness.

Join relationship

Includes basically all relationships in the data layer except inheritance.

Context	Check	Remarks
Join relationship	Every join relation must have a (source | target) alias.	Aliases must have a length of at least two chars. Recommended are at least four chars.
Join relationship	Relationship source aliases must be unique for all incoming relations of an entity.	
Join relationship	Relationship target aliases must be unique for all outgoing relations of an entity.	
Join relationship	The (sourceField | targetField) attribute must contain either 'id' for automatic primary key or a comma separated list of (source | target) fields to be referenced. Note that the separator is ', ' including the space char.	
Join relationship	The amount of join columns must be equal for association source and target sides.	If source or target field are not *id* ModuleStudio splits both values by the separator above and compares the amount of elements on both sides. This enables joins over multiple fields.
Join relationship	The field *sourceField* | *targetField* must be an integer field with a length of 11.	Is only checked if the (source | target) field is named *id*.
Join relationship	The fields *sourceField* and *targetField* must have the same type and length values.	Is only checked if the source and target fields are both not named *id*.
Join relationship	Self relation must not reference the *sourceField* field for both source and target.	For self relation the source field must not be equal to the target field (as the database needs two fields in order to store both sides).
Join relationship	Between two entities there must not be multiple join relations with cascade options.	

One to one relationship

Context	Check	Remarks
One to one relationship	The edit type ACTIVE_CHOOSE_PASSIVE_NONE is only valid for many to many relationships.	
One to one relationship	The edit type ACTIVE_EDIT_PASSIVE_NONE is only valid for many to many relationships.	
One to one relationship	Entities with soft-deleteable need a remove cascade for any incoming one-to-one relationships.	See this issue for more information.

Many to one relationship

Context	Check	Remarks
Many to one relationship	The edit type ACTIVE_CHOOSE_PASSIVE_NONE is only valid for many to many relationships.	
Many to one relationship	The edit type ACTIVE_EDIT_PASSIVE_NONE is only valid for many to many relationships.	
Many to one relationship	A many-to-one relation must not be bidirectional.	

One to many relationship

Context	Check	Remarks
One to many relationship	The edit type ACTIVE_CHOOSE_PASSIVE_NONE is only valid for many to many relationships.	
One to many relationship	The edit type ACTIVE_EDIT_PASSIVE_NONE is only valid for many to many relationships.	
One to many relationship	The indexBy attribute points to an invalid field of the target entity.	Checks whether the target entity contains a field equally named like the *indexBy* property.
One to many relationship	IndexBy fields must be unique.	The target entity field used for *indexBy* must have *unique* validator enabled.

Many to many relationship

Context	Check	Remarks
Many to many relationship	The indexBy attribute points to an invalid field of the target entity.	Checks whether the target entity contains a field equally named like the *indexBy* property.
Many to many relationship	IndexBy fields must be unique.	The target entity field used for *indexBy* must have *unique* validator enabled.
Many to many relationship	A many-to-many relation must have the refClass attribute defined.	The *reference class* defines the name of the entity managing the many to many relationship. If for example many *persons* have many *addresses*, you could choose *personAddress* as reference class.
Many to many relationship	The refClass attribute must be unique for all relations.	
Many to many relationship	The refClass attribute must not be equal to any entity name.	The generator creates additional entity classes so unique naming is ensured to prevent naming collisions.

Inheritance relationship

Context	Check	Remarks
Inheritance relationship	Self relations are not allowed for inheritance relationships.	
Inheritance relationship	The discriminatorColumn attribute must not be empty for inheritance relations.	Per default this attribute has the value *discr* so this shouldn't happen as long as you don't delete this value. ModuleStudio uses this value to tell Doctrine 2 where to store the type of stored entities.
Inheritance relationship	Please rename the *discriminatorColumn* field as it is reserved for the discriminator column, or change corresponding attribute.	Happens if the target entity includes a field with the same name as specified in the *discriminator setting*.
Model container with inheritance relationships	Inheritance cycles are not allowed.	Recursive check to detect cycles within inheritance hierarchies.

5.4.5. Entity index

Context	Check	Remarks
Entity index	Index must be assigned to an entity.	Should not occur in practice, this is just for completeness.
Entity index	Every index must have a name.	The index name must have a length of at least two chars. Recommended are at least four chars.
Entity index	Every index must contain at least one item referencing an entity field.	
Entity index	The length of all index fields must not be higher than 333.	The limit is 1000 bytes, while UTF-8 requires three bytes for each char.
Entity index	Index names must be unique per entity.	
Entity index	Index item names must be unique per index.	
Entity index item	The index item must be assigned to an index.	Should not occur in practice, this is just for completeness.
Entity index item	Every index item must have the same name as the referenced entity field.	This occurs if no equally named field is found in the entity.

5.4.6. Variables

Context	Check	Remarks
Variable container	Every var container must have a name.	
Model container with variable containers	Var container names must be unique.	
Model container with variable containers	Var container sort positions must be unique.	
Variable container	Every var container must contain at least one variable.	
Variable container	Var container must be assigned to a model container.	Should not occur in practice, this is just for completeness.
Variable	The var must be assigned to a container.	Should not occur in practice, this is just for completeness.
Variable	Every var must have a name.	The var name must have a length of at least two chars. Recommended are at least four chars.
Variable	A var must not have the same name as an entity.	
List var	The list must contain at least one item.	

5.4.7. Other elements

Context	Check	Remarks
Calculated field	The field must rely on at least one derived field.	Not relevant yet, this is for future.
Entity event listener	The listener must contain at least one operation.	Not relevant yet, this is for future.
Transform object	Every transformation must be assigned to an event listener.	Not relevant yet, this is for future. Should not occur in practice, this is just for completeness.

5.5. Controller layer

5.5.1. Controller container

Context	Check	Remarks
Controller container	The controller container must contain at least one controller.	
Controller container with controllers	You must have an admin controller for hook subscriber functionality.	Since Zikula 1.3 it is highly recommended to have an admin area as there is an additional page for defining hook assignments for different areas.
Controller container with controllers	Controller names must be unique.	For example there may not be two controllers which are both named *admin*.
Controller container	Please create an ajax controller including an arbitrary action, like main, because it is required for processing your model properly.	This happens if your model layer contains relationships or an entity with either user fields or tree extension. As the generator creates additonal ajax methods in these cases it requires an ajax controller for keeping this information also in the model for documentation.

5.5.2. Controller

Context	Check	Remarks
Controller	The controller must be assigned to a controller container.	Should not occur in practice, this is just for completeness.
Controller	Every controller must have a name.	Controller name must have a length of at least three chars. Should be at least four chars.
Custom controller	Controller name must not contain underscores.	Underscores are not allowed as they are used for class autoloading.
Custom controller	Controller name must not be Admin, User, Account, Ajax, Search or Init.	These are reserved names and may therefore not be used for custom controllers.
Controller	Every controller must contain at least one action.	
Controller with actions	There must not exist more than one (main | view | display | edit | delete) action in one controller.	
Controller with actions	Names of custom states in one controller must be unique.	
Controller with actions	This controller may not contain a delete action. Please remove it.	Appears for special controllers, like ajax. Those are reserved for special purposes and not intended to be called by the user as normal controllers.

5.5.3. Action

Context	Check	Remarks
Action	The action must be assigned to a controller.	Should not occur in practice, this is just for completeness.
Custom action	The action must have a name.	Action name must not be *New* or *Hooks*. Must have a length of at least four chars, whereby at least six chars are recommended.
Custom action	Action name must not be Main, View, Display, Edit or Delete.	These are reserved names and may therefore not be used for custom actions.
Action	Every action must be assigned to an action handler.	Not active yet as the controller editor needs some additional features first.

5.5.4. Action handler

Context	Check	Remarks
Action handler	Every action handler must have a name.	Must have a length of at least four chars, whereby at least six chars are recommended.
Action handler	Action handler name must not be List, Detail or Edit.	These are reserved names and may therefore not be used for custom handlers.
Action handler	The action handler must contain at least one action events.	
Action handler with events	The action handler must contain at least one start event.	There must not exist more than one start event in one action handler.
Action handler	Every action handler must have the same name as the main entity it refers to (e.g. *CustomerHandler*).	Not active yet as the controller editor needs some additional features first.

5.5.5. Action event

Context	Check	Remarks
Action event	Action event must be assigned to an action handler.	Should not occur in practice, this is just for completeness.
Action event	Every action event must have a name.	Must have a length of at least four chars, whereby at least six chars are recommended.

5.5.6. Transition

Context	Check	Remarks
Transition	Transition must be assigned to a controller container.	Should not occur in practice, this is just for completeness. Inactive at the moment anyway.
Transition	Every transition must have a name.	Must have a length of at least two chars, whereby at least four chars are recommended.
Transition	Every transition must have a (source | destination) action.	A transition can not link to itself (this is inactive at the moment though).
Controller container with transitions	There must not exist more than one transition between two actions.	

5.6. View layer

5.6.1. View container

Context	Check	Remarks
View container	The view container must contain at least one view.	This rule is currently not active as the view editor is not ready yet.
View container with views	View names must be unique.	For example there may not be two views which are both named *display*.

5.6.2. View

Context	Check	Remarks
View	Every view must have a name.	View name must be longer than at least three chars. Should be more than five chars.
View	Every view must have a root panel.	
View	Every view must have the same name as the main entity it's controller refers to (e.g. *CustomerView*).	

5.6.3. Root panel

Context	Check	Remarks
Root panel	Every root panel must have a presenter.	
Root panel	Every root panel must contain at least one composite container.	
Root panel	Every view must contain at least one submit activator.	Ideally every view should contain also at least one cancel activator.
Root panel	Check	

5.6.4. Other UI elements

Context	Check	Remarks
Composite container	Composite container must be assigned to a root panel.	
Tabbed pane	A tabbed pane must contain at least two child containers.	
Sub container	Child container must be assigned to a parent container.	
Sub container	Every sub container must contain at least one field.	
Field	Field must be assigned to a container.	
Form label	Label must point to a field.	

5.6.5. Layout order

Context	Check	Remarks
Layout order	Layout order must be assigned to a presentation container.	
Layout order	Every transition must have a (source | destination) container.	A layout order can not link to itself.
View container with layout orders	There must not exist more than one layout order between two containers.	

5.7. Workflow layer

The workflow layer is not implemented yet. This section is just a dummy for future.

5.8. Additional notes

None yet.

6. Application generator

6.1. Introduction

The main use case for ModuleStudio application models is the creation of Zikula extensions. This section describes how the generator works and which artifacts are created from which model elements.

ModuleStudio brings also other generator cartridges (§8.1) for creating further information with regards to documentation and reporting tasks.

6.2. Basic idea

Every application consists of different types of code parts. While some code is unique for each application, most parts can be derived from that and is therefore very similar for a whole software systems family. Those code parts are known as *boilerplate code*.

A very simple example will make this clear very quickly:

```
/**
 * imagine some long comments about this class here
 * @ORM\Entity ...
 * some more annotations
 */
class MyModule_Entity_Person extends Zikula_EntityAccess
{
    /**
     * imagine some long comments about this field here
     * @ORM\Column ...
     * some more annotations
     */
    protected $firstName;
     /**
     * imagine some long comments about this field here
     * @ORM\Column ...
     * some more annotations
     */
    protected $lastName;
}
```

This code has obviously not very much knowledge which is essential for this certain application. Reduced to what is really required from a functional view one would get something like:

```
entity person {
    string firstName
    string lastName
}
```

Thought a little further the generator helps reaching a constantly high code quality, as all implementation details are always considered completely. For example if a new extension is activated for an entity this is not forgotten anywhere inside the code.

6.3. How it works

There is a tutorial showing how calling the generator within ModuleStudio looks like: Using the generator. The only requirement is that you have opened your application model and there are no validation errors (§5) left.

6.3.1. Input dialogs

First you have to select which generator cartridges you want to execute.

After that you have to select which reports you want to create (only if you did not unselect the *reporting* cartridge before).

6.4. The workflow

The rough steps of the generator workflow are as follows:

1. Ask for input parameters (for example desired output folder, cartridges and reports).
2. Empty output directory.
3. Export dumps of editor diagrams (if reporting cartridge is selected).
4. Read input model.
5. Perform validation.
6. Transformation to add primary and foreign key fields.
7. Call generator inner workflow for each selected cartridge.

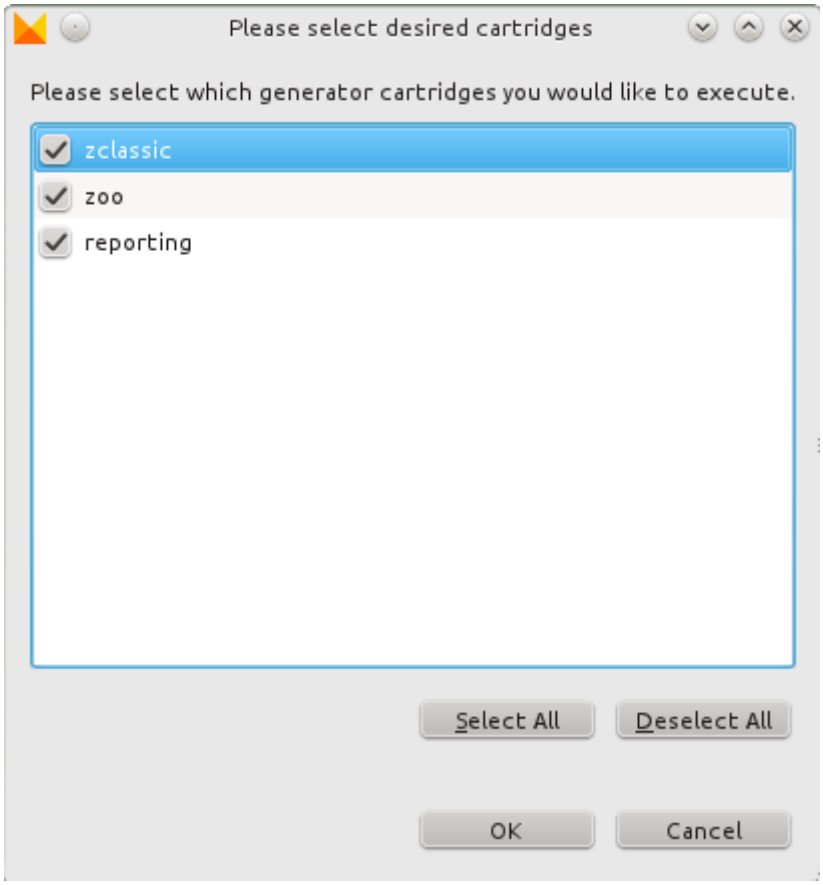

Figure 6.1.: Cartridge selection

6.5. Generator reference

This reference goes through all available model elements as well as their properties, describing what the generator does with this information and which things are still missing in the created implementation.

You will notice that there are also some elements included which are not showing up in ModuleStudio itself yet. This is for showing up the picture we have in mind when designing the modeling language as a whole. In most cases where the generator is not flexible yet this is caused by limitations in the current editors.

Most screenshots in this section are taken from the example application called *Recipe-Manager*.

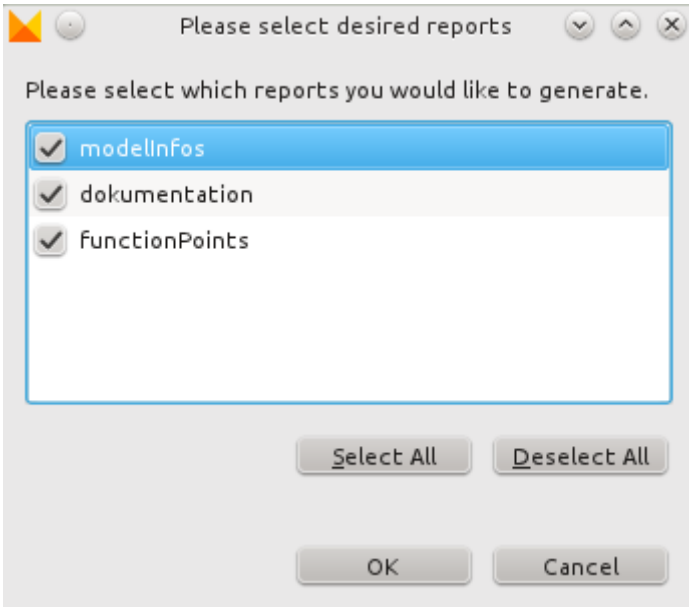

Figure 6.2.: Report selection

6.5.1. Application layer

Language elements

Named object

This is the common base class of almost all model elements.
It includes the following properties:

- *name* - The name of the element.
- *documentation* - A description for documenting the element.

If a *documentation* is defined for an entity this will be shown right after the heading of the corresponding view template. So you could for example add a description for the *person* entity explaining what persons are and what information they store. If a user does then what the persons list he knows immediately what he is looking at.

Application

Represents an application described by the model.
It includes the following basic properties:

- *vendor* - The vendor of the application. Usually this is the name of a company or institution. Becomes important in a future version as the vendor and name of an application are going to be combined to a unique name. Then it is possible to have for example multiple News modules installed from different vendors.

- *author* - The author of the application. Usually this is the full name of the developer.
- *email* - The email address of the developer.
- *license* - The license of this application. Defaults to LGPL. If either GPL or LGPL are used the generator creates corresponding license files, too.
- *url* - The homepage of the developer.
- *version* - The application version. Must conform to the pattern *x.y.z* - for example *1.0.0* which is also the default value. Will be used in the version class of the created extension.

These basic fields are mainly used by the generator to create a meaningful file header. An application has some more fields for specifying specific aspects:

- *applicationType* - The kind of application described by the model. See below (§6.5.1).
- *targetCoreVersion* - The targeted Zikula core version. See below (§6.5.1). Note this setting is obsolete on application level. Please use the settings container (§6.5.1) property instead.
- *interactiveInstallation* - A boolean specifying whether an interactive installation should be used or not. The default value is *false* which lets the generator create a normal installer without any required user input. If you set it to true it will additionally generate some init templates as well as a corresponding controller for the interactive installer. *Important note:* support for interactive installers is currently disabled as this topic is broken and being overhauled in the Zikula core.
- *modelPath* - Physical file path to the application model. This is obsolete and will be removed in a later version. Please keep it empty.
- *prefix* - A prefix for all database tables of this application. Will be used in entity classes.

An application may furthermore have the following references:

- *controllers* - Allows referencing one or more controller layers (§6.5.1).
- *generatorSettings* - Allows specifying desired generator features and behaviour. More details in the settings container (§6.5.1) section.
- *models* - Allows referencing one or more model layers (§6.5.1).
- *referredApplications* - Allows referencing other applications. See below (§6.5.1).
- *views* - Allows referencing one or more view layers (§6.5.1).

Figure 6.3.: Application properties

Application type

The kind of application described by the model.
Can be one of the following options:

- *WEB* - Web applications. This is the default value.
- *NATIVE* - Native applications.
- *EMBEDDED* - Embedded applications.

At the moment this value is completely ignored by the generator. You should nevertheless keep it set to *WEB* to be on the safe side for future.

Core version

Zikula version for which the application should be generated.
Can be one of the following options:

- *ZKPRE14* - Aims on Zikula 1.4.0 and later. This is the default value and useful when developing for future.
- *ZK136* - For Zikula 1.3.7 and earlier. This is for backwards compatibility and useful when maintaining extensions in production.
- *ZK135* - For Zikula 1.3.7 and earlier. This is for backwards compatibility and useful when maintaining extensions in production.

Settings container

Each application may contain one settings container for customising generator settings. You can control which features should be generated and take influence on some behavioural aspects of the generator.

A settings container has the following fields:

- *targetCoreVersion* - The targeted Zikula core version. See above (§6.5.1).
- *isSystemModule* - A boolean specifying whether the model describes a system module or not. Default value is *false*.
- *generateAccountApi* - A boolean specifying whether account panel integration should be generated or not. Default value is *true*.
- *generateSearchApi* - A boolean specifying whether search integration should be generated or not. Requires at least one string or text field in any entity. Default value is *true*.
- *generateMailzApi* - A boolean specifying whether Mailz support should be generated or not. Default value is *true*.
- *generateListBlock* - A boolean specifying whether a generic list block should be generated or not. Default value is *true*.
- *generateModerationBlock* - A boolean specifying whether a moderation block should be generated or not. Requires at least one entity with a workflow including approval. Default value is *true*.
- *generateListContentType* - A boolean specifying whether a content type for collection lists should be generated or not. Default value is *true*.
- *generateDetailContentType* - A boolean specifying whether a content type for single objects should be generated or not. Requires user controller containing a display action. Default value is *true*.
- *generateNewsletterPlugin* - A boolean specifying whether a Newsletter plugin should be generated or not. Default value is *true*.
- *generateModerationPanel* - A boolean specifying whether a moderation panel should be generated or not. Requires at least one entity with a workflow including approval. Default value is *true*.
- *generatePendingContentSupport* - A boolean specifying whether support for pending content should be generated or not. Requires at least one entity with a workflow including approval. Default value is *true*.
- *generateExternalControllerAndFinder* - A boolean specifying whether a controller for external calls providing a generic finder component should be generated or not. Default value is *true*.

- *generateScribitePlugins* - A boolean specifying whether support for several Scribite editors should be generated or not. Requires external controller with finder component. Default value is *true*.
- *generateTagSupport* - A boolean specifying whether tag support should be generated or not. Requires user or admin controller containing a display action. Default value is *true*.
- *generateRssTemplates* - A boolean specifying whether rss view templates should be generated or not. Default value is *true*.
- *generateAtomTemplates* - A boolean specifying whether atom view templates should be generated or not. Default value is *true*.
- *generateCsvTemplates* - A boolean specifying whether csv view templates should be generated or not. Default value is *true*.
- *generateXmlTemplates* - A boolean specifying whether xml display and view templates should be generated or not. Default value is *true*.
- *generateJsonTemplates* - A boolean specifying whether json templates should be generated or not. Default value is *true*.
- *generateKmlTemplates* - A boolean specifying whether kml templates should be generated or not. Requires geographical flag on corresponding entities. Default value is *true*.
- *generateIcsTemplates* - A boolean specifying whether ics (iCalendar) templates should be generated or not. Requires start date and end date fields on corresponding entities. Default value is *true*.
- *generateOnlyBaseClasses* - A boolean specifying whether only base classes should be generated. May be useful for doing simple upgrades without structural changes. Default value is *false*.
- *skipFiles* - Comma-separated blacklist with each entry representing a file which should not be generated. Default value is an empty string.
- *markFiles* - Comma-separated list with file pathes which should be marked by special file names. Instead of the original name each file is generated using the pattern *filename.generated.extension*. This setting can be useful for doing bigger merges comparing the generated version with a customised one. Default value is an empty string.
- *timestampAllGeneratedFiles* - A boolean specifying whether the generated by message should contain a timestamp in all files or only in the Version class. Default value is *false*.
- *generatePoweredByBacklinksIntoFooterTemplates* - A boolean specifying whether generated footer templates should contain backlinks to the ModuleStudio homepage. Default value is *true*.

- *generateTests* - A boolean specifying whether test cases should be generated or not. Default value is *true*.
- *writeModelToDocs* - A boolean specifying whether the model files are written into the module's docs folder or into a separate folder outside the module. Default value is *false*.

Referred application

Represents an application whose model file is being imported (e.g. to reference other entities or other extensions which are incorporated by api calls).

An application reference has the following fields:

- *minimumVersion* - The minimum version this reference applies for.
- *maximumVersion* - The maximum version this reference applies for.
- *importURI* - URI to imported model file.
- *dependencyType* - The type of dependency which should be used for the referred application. See below (§6.5.1).

Application dependency type

Specifies the kind of dependency to a certain application.

Can be one of the following options:

- *REQUIREMENT* - The module is required, for example to join related entities.
- *RECOMMENDATION* - The module is recommended, for example to provide enhanced integration functionality.
- *CONFLICT* - The module is in conflict with the modeled one, for example due to overlapping functionality.

The generator uses this value in the corresponding module dependency created in the version class.

Model container

Container class for carrying elements in the model layer.

A model container may have the following references:

- *application* - Reference to the owning element.
- *defaultDataSource* - Whether this container represents the default data source or not. The default value is *true* which can only be assigned to one data source which is treated like an internal Zikula storage. Additional containers are processed as external data sources.

- *entities* - Allows referencing one or more entities (§6.5.2).
- *numExampleRows* - The amount of example rows to create for entities in this model layer. Default value is 0. Note that if you activate the *categorisable* property for an entity the generated installer relies on that you did not remove the default categories of Zikula. If you deleted them please set the amount of example rows to 0 to avoid problems.
- *relations* - Allows referencing one or more relationships (§6.5.2).
- *variables* - Allows referencing one or more variables (§6.5.2).

For each entity as well as for many to many relationships according classes are generated. More details are explained in the model layer section (§6.5.2). However external data sources are not treated correctly (see #5 for more information).

Controller container

Container class for carrying elements in the controller layer.
A controller container may have the following references:

- *application* - Reference to the owning element.
- *controllers* - Allows referencing one or more controllers (§6.5.3).
- *handlers* - Allows referencing one or more action handlers (§6.5.3).
- *modelContext* - Allows referencing one or more model containers (§6.5.1).
- *processViews* - Allows referencing a view container (§6.5.1).
- *transitions* - Allows referencing one or more transitions (§6.5.3).

As all the container elements also the controller container is primarily a composite element containing a sublayer. The generator creates according controller classes, but does not separate different controller containers on code level.

View container

Container class for carrying elements in the view layer.
A view container may have the following references:

- *application* - Reference to the owning element.
- *controller* - Allows referencing a controller container (§6.5.1).
- *layoutOrders* - Allows referencing one or more layout orders (§6.5.4).
- *views* - Allows referencing one or more views (§6.5.4).

As there is no view editor available yet this element is not relevant for the generator yet. Instead it only creates default templates for each existing controller action (§6.5.3) and entity (§6.5.2), as well as some common templates which are included or required for some extensions.

Combinations and edge cases

Multiple containers may not be considered properly in all implementation areas yet. There are some expressions which must be rechecked in order to limit some generation parts to certain container element instances only. So it can happen for example that several data layer artifacts from additional data sources are treated like if they were part of the primary internal one.

Therefore it can also be that the relations between your containers are not reflected as you would expect it. To some degree this can also be a result of missing dependencies, like for example missing editor capabilities to design the view areas.

So for the moment multiple containers are often treated like if they were one big merged container. With time this will change though, as more and more expressions become more explicite including or rejecting certain model elements more precisely.

6.5.2. Model layer

The model layer in ModuleStudio has been designed for a precise description of entities and associations. To understand all the elements and properties please read the Doctrine 2 documentation before.

Language elements

Entity

Represents an entity in the data layer which is mapped to a database table.

It includes the following properties:

- *attributable* - A boolean specifying whether this entity should have attributes or not. If set to *true* the generator creates an additional entity for managing the attributes. During edit (§6.5.3) actions it is possible to input values for three predefined attributes. These will also be shown again on display (§6.5.3) pages. There is no included support yet for arbitrary attributes like they are known from the Categories administration area. While the display side is ready for that, the edit page needs some dynamic support for creating new attributes on the fly.

- *categorisable* - A boolean specifying whether this entity should have categories or not. If set to *true* the generator creates an additional entity for managing the categories. During edit (§6.5.3) actions it is possible to select a desired category. This category will also be shown again on display (§6.5.3) pages and in quick navigation forms of view (§6.5.3) pages. Generated applications support filtering by categories as well as multiple category registries / properties / trees, however the implementation uses only *Main* per default. There is no built-in permission scheme based on categories implemented yet. Note that if you activate the *categorisable* property for an entity the generated installer relies on that you did not remove the default categories of Zikula. If you deleted them please set the amount of example rows to 0 to avoid problems.
- *categorisableMultiSelection* - A boolean specifying whether multiple categories can be selected or not.
- *changeTrackingPolicy* - How change detection is being done (see below (§6.5.2)). The default value is *DEFERRED_IMPLICIT*.
- *displayPattern* - Pattern for displaying instances of this entity. In earlier ModuleStudio versions one had to mark one entity field as *leading*. However, this was not flexible enough in practice. With the display pattern you can specify arbitrary expressions which are used as textual representation for instances of this entity. For most cases you may want so declare just one field, which is done like *#title#*. A more complex example would be *#lastName#, #firstName# (#age# years)*. Of course all fields must exist in the entity with exactly the names used within the display pattern.
- *geographical* - A boolean specifying whether the geographical extension is used or not. If set to *true* the generator will create two additional fields named *latitude* and *longitude*. Also it will consider them in all important application areas and provide an export for the kml format. During the creation of a new entity with geographical support a nice geolocation feature is used to ask the user for his current location. Also there is an included integration of the Mapstraction class allowing you to use different map providers in your application.
- *hasArchive* - Whether the workflow should include an archived state with automatic archiving. Requires a datetime (§6.5.2) or date (§6.5.2) field which has been designated as end date. See workflow types (§6.5.2) for more information. The default value is *false*.
- *hasTray* - Whether the workflow should include a suspended state. See workflow types (§6.5.2) for more information. The default value is *false*.
- *identifierStrategy* - Whether and which identifier strategy (§6.5.2) is applied. The default value is *NONE*.
- *leading* - A boolean specifying whether this is the primary (and default) entity or not.
- *lockType* - Whether and which locking strategy (§6.5.2) is applied.

- *loggable* - A boolean specifying whether the loggable behavior is used or not. The generator will create an additional entity for managing the log entries if set to *true*. There is no user interface for the version management yet (see #30 for more information).
- *mappedSuperClass* - A boolean specifying this is a mapped superclass or a normal entity. A mapped super class is not able to store data. At the moment the generator creates many unrequired things for mapped super classes though, like for example view templates. The only thing where mapped super classes are already rejected correctly are repository classes.
- *metaData* - A boolean specifying whether this entity should have support for meta data. If set to *true* the generator creates additional inclusion templates for displaying and changing corresponding fields.
- *nameMultiple* - Plural form of the name. The generator uses this for collections, list views and other areas where multiple entities are used.
- *ownerPermission* - Whether users should be able to manage and edit their own data. Defines also whether the workflow should include a deferred state. See workflow types (§6.5.2) for more information. The default value is *false*.
- *readOnly* - A boolean specifying whether this entity is read only or not. If set to *true* editing will not be possible.
- *slugLength* - Length of slug field. Defaults to 255. An entity is sluggable as soon as at least one of its fields set *sluggable position* to a value greater than 0.
- *slugSeparator* - Separator which will separate words in slug. Default value is - like in Zikula, too.
- *slugStyle* - Which slug style (§6.5.2) is used.
- *slugUnique* - A boolean specifying if the slug is unique or not. Default value is *true*.
- *slugUpdatable* - A boolean specifying if the slug can be changed or not. Default value is *true*.
- *softDeleteable* - Whether deleted items should only be marked as deleted instead of deleting them. Defines also whether the entity workflow provides means for trashing and recovering items or for deleting them. See workflow types (§6.5.2) for more information. The default value is *false*.
- *tree* - Whether and which tree strategy is applied. More information about what the generator creates for trees can be found in the the section about entity tree types (§6.5.2).
- *standardFields* - A boolean specifying whether the standard fields extension is used or not. If set to *true* the entity will get four additional fields for storing the id of the user who created the item, the id of the user who did the last update, as well as the creation and update dates. This information will be included on display (§6.5.3) and edit (§6.5.3) actions.

- *workflow* - The workflow which is applied to this entity. See workflow types (§6.5.2) for more information. The default value is *NONE*.
- *onAccountDeletionCreator* - Controls how an app should change the creator when users are deleted. Only relevant if *standardFields* is enabled. Default value is *ADMIN*. The available options are listed here (§6.5.2).
- *onAccountDeletionLastEditor* - Controls how an app should change the last editor when users are deleted. Only relevant if *standardFields* is enabled. Default value is *ADMIN*. The available options are listed here (§6.5.2).

Normally all created classes are generated twice. Thereby an empty concrete class inherits from an abstract base class containing the whole generator code. The motivation behind this separation is that your own code keeps free from generated artifacts.

Example for Zikula 1.4.0:

```
namespace MyModule\Entity\Validator\Base;
 class Person extends \MyModule\Validator
{
    // generator code
}
 namespace MyModule\Entity\Validator;
 use MyModule\Entity\Validator\Base\Person as BasePerson;
 class Person extends BasePerson
{
    // manual code
}
```

Example for Zikula 1.3.7 and below:

```
class MyModule_Entity_Validator_Base_Person
    extends MyModule_Validator
{
    // generator code
}
 class MyModule_Entity_Validator_Person
    extends MyModule_Entity_Validator_Base_Person
{
    // manual code
}
```

One exception for this scheme is inheritance (§6.5.2).

Please note that arbitrary *filtering is not possible at the moment* until FilterUtil has been upgraded to support Doctrine 2.

Whenever you want to change the default implementation you can add corresponding extensions. If you recognise that you are doing the same changes again and again please submit them as patches for the generator.

An entity may have the following references:

- *container* - Reference to the owning element.
- *fields* - Allows referencing one or more entity fields (§6.5.2).
- *incoming* - Allows referencing one or more incoming relationships (§6.5.2).
- *indexes* - Allows referencing one or more indexes (§6.5.2).
- *listeners* - Allows referencing one or more event listeners (§6.5.2).
- *outgoing* - Allows referencing one or more outgoing relationships (§6.5.2).

One additional note about slugs and permalinks: the generated short url handlers will understand different url schemes for the display pages (§6.5.3) depending on the entity settings.

- Entities which are not sluggable use the identifier for display urls, for example *mymodule/person/5.html*.
- Entities with unique slugs use the slug for display urls, for example *mymodule/person/william-smith.html*.
- Entities with non-unique slugs combine both methods, for example *mymodule/person/william-smith.5.html*.

Entity field

Represents an entity field in the data layer.

This base class has the following children at the moment:

- Derived fields (§6.5.2) correspond to normal columns which are stored in a database.
- Calculated fields (§6.5.2) correspond to fields which can calculate their values based on other fields.

An entity field may have the following references:

- *entity* - Reference to the owning element.

Derived field

Represents an entity field in the data layer which is mapped to a database column. A derived field comes straight from the data source.

A derived field has the following properties in addition to the common entity field (§6.5.2) settings:

- *dbName* - Name of the database column represented by this field. The default value is an empty string which means that the field name is used for the column, too. This property is primarily interesting for avoiding changes in the database schema when migrating legacy apps.
- *defaultValue* - The default value of the field. This default value is used when creating new entities with the edit action (§6.5.3).
- *leading* - A boolean specifying whether this is the primary (and default) field in this entity. Default value is *false*. This setting is obsolete. Use the *displayPattern* property of the containing entity (§6.5.2) instead.
- *mandatory* - A boolean specifying whether this field is mandatory or not. The default value is *true*.
- *nullable* - A boolean specifying whether the field may be null or not. The default value is *false*. A nullable field may not be mandatory at the same time.
- *primaryKey* - A boolean specifying whether this is a primary key field or not. Default value is *false*. Usually there is no need to enable this for any fields as the generator adds primary and foreign key fields automatically. The only use case where the manual definition of primary keys makes sense is having composite keys. This should work in general with regards to the generated data layer, but support on controller and view layers in the created application may not be prepared properly yet for that.
- *readonly* - A boolean specifying whether this a read only field or not. The default value is *false*. If set to *true* then this field may not be changed during editing.
- *sluggablePosition* - Position of this field in the created slugs. A value of 0 means that this field is not part of the slug at all. If at least one field in an entity has a sluggable position greater than 0 then this entity is considered as sluggable. In this case a permalink is built automatically from all fields in ascending position. See the slug properties on entity level (§6.5.2) for slug-related configuration options.
- *sortableGroup* - A boolean specifying whether this field acts as grouping criteria for the sortable extension. The default value is *false*. SortableGroup is not fully implemented yet, do not use if you not understand the function.
- *translatable* - A boolean specifying whether this field is translatable or not. The default value is *false*. If at least one field in an entity is translatable the generator creates an additional class for managing the translation entities. Overall support for translations in the application should get you started.

- *unique* - A boolean specifying whether this field is unique or not. The default value is *false*. If set to *true* then an additional validator cares for enforcing the unique constraint on client and server level.

All fields are implemented as entity class member vars. The following sections will look at the different field types in detail.

Calculated field

Represents an entity field which can dynamically compute it's value based on other fields.

A calculated field may have the following references in addition to the common entity field (§6.5.2) settings:

- *operands* - Allows referencing one or more derived fields (§6.5.2).

Calculated fields are not part of the model editor yet and will therefore be ignored by the generator.

Boolean field

Represents a field type for storing boolean values.

A boolean field has the following properties in addition to the common entity field (§6.5.2) settings:

- *ajaxTogglability* - Boolean indicating whether it is possible to switch this flag with ajax or not. If set to *true* all view and display pages will contain corresponding links instead of only simple state images.

The generator will treat boolean values as checkbox input elements in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *yesno* modifier is used to show an image indicating the boolean value (green check or red cross).

Abstract integer field

Represents an abstract integer field for grouping different implementations of this field type.

An abstract integer field has the following properties in addition to the common entity field (§6.5.2) settings:

- *length* - The length of this field. This controls whether the Doctrine mapping type will be *integer*, *bigint* or *smallint*. Default value is 11.
- *sortablePosition* - A boolean specifying whether this field stores the position for the sortable extension or not. If set to *true* this field will be used as default sorting criteria. There is no built-in reordering possibility, for example with drag n drop, implemented yet (see #29 for more information).

Integer field

Represents a field type for storing integer numbers.

An integer field has the following properties in addition to the common abstract integer field (§6.5.2) settings:

- *aggregateFor* - Aggregate field: one-to-many target alias and field name (syntax: *views#amount*) which causes the generator creating special methods for aggregation.
- *maxValue* - Maximal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *minValue* - Minimal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *version* - A boolean specifying whether this field should act as a version. If set to *true* the owning entity will need to use optimistic locking (§6.5.2). There is no user interface for version management generated yet.

In edit (§6.5.3) pages the generator will use int input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) templates the value will just be shown.

Decimal field

Represents a field type for storing decimal numbers.

A decimal field has the following properties in addition to the common entity field (§6.5.2) settings:

- *aggregationField* - A boolean specifying whether this field should act as an aggregate field. If set to *true* the generator creates special methods for aggregation.
- *currency* - A boolean specifying whether this field should be treated as currency. If set to *true* the generator will use the *formatcurrency* modifier instead of *formatnumber* during output.
- *length* - The length of this field. Default value is 10.
- *maxValue* - Maximal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *minValue* - Minimal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *scale* - The amount of digits after the dot. Default value is 2.

In edit (§6.5.3) pages the generator will use float input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) templates the value will just be shown using the *formatnumber* modifier.

Float field

Represents a field type for storing float numbers.

A float field has the following properties in addition to the common entity field (§6.5.2) settings:

- *aggregationField* - A boolean specifying whether this field should act as an aggregate field. If set to *true* the generator creates special methods for aggregation.
- *currency* - A boolean specifying whether this field should be treated as currency. If set to *true* the generator will use the *formatcurrency* modifier instead of *formatnumber* during output.
- *length* - The length of this field. Default value is 10.
- *maxValue* - Maximal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *minValue* - Minimal value. If set to a value other than 0 then a validator will enforce this constraint on client and server side.

In edit (§6.5.3) pages the generator will use float input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) pages the value will just be shown using the *formatnumber* modifier.

Abstract string field

Represents an abstract string field for grouping different implementations of this field type.

An abstract string field has the following properties in addition to the common entity field (§6.5.2) settings:

- *fixed* - A boolean specifying whether this field has a fixed length or not.
- *minLength* - Minimal length. If set to a value other than 0 then a validator will enforce this constraint on client and server side.
- *nospace* - A boolean specifying whether space chars are forbidden or not.
- *regexp* - Regular expression to validate against. Possible example values are $/\hat{}\backslash w+/$ and $/\backslash d/$.
- *regexpOpposite* - A boolean specifying the logical way of checking *regexp*. If set to *true* then validation will pass only if the given string does not match the pattern.

If one of these properties is set to *true* a corresponding validator will check this constraint on client and server level.

String field

Represents a field type for storing string values.

A string field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *country* - A boolean specifying whether this field represents a country code or not. If set to *true* a country selector is used in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates an output modifier is used to display the full country name instead of the unreadable country code.
- *creditCard* - A boolean specify whether this field represents a credit card number or not. Only applicable for the 1.4 target core version.
- *currency* - A boolean specify whether this field represents a 3-letter ISO 4217 currency name or not. Possible example values are *USD* or *EUR*. Only applicable for the 1.4 target core version. In forms it will be rendered as a text input, until Symfony Forms (which is offering a dedicated field type) is used for the 1.4 target.
- *htmlcolour* - A boolean specifying whether this field represents a html color code (like #003399) or not. If set to *true* a colour picker is used in edit (§6.5.3) pages for convenient selection of colour codes.
- *iban* - A boolean specify whether this field represents a bank account number in IBAN (International Bank Account Number) format. Only applicable for the 1.4 target core version.
- *isbn* - Allows to define whether this field represents a number in ISBN (International Standard Book Number). You can choose corresponding validation options (§6.5.2). covered ip address scope (§6.5.2). Only applicable for the 1.4 target core version.
- *issn* - Allows to define whether this field represents a number in ISSN (International Standard Serial Number). You can choose corresponding validation options (§6.5.2). covered ip address scope (§6.5.2). Only applicable for the 1.4 target core version.
- *ipAddress* - Allows to define whether this field represents an IP address. You can choose the covered ip address scope (§6.5.2). Only applicable for the 1.4 target core version.
- *language* - A boolean specifying whether this field represents an Unicode language identifier or not. Possible example values are *fr* or *zh-Hant*. If set to *true* a language selector is used in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *getlanguagename* modifier is used to display the full name instead of the unreadable language code.
- *length* - The length of this field. Default value is 255.

- *locale* - A boolean specify whether this field represents a locale or not. Possible example values are *fr* or *fr_FR*. In forms it will be rendered as a language selector, until Symfony Forms (which is offering different field types) is used for the 1.4 target.
- *password* - A boolean specifying whether this field represents a password or not. If set to *true* a password input element will be used instead of a normal one in edit (§6.5.3) pages.
- *uuid* - A boolean specifying whether this field represents an UUID (Universally Unique Identifier) or not.

In edit (§6.5.3) pages the generator will use singleline input elements for string fields - except you defined something else (like language or password). Other validations are added together and applied as well.

ISBN style

Represents different ISBN formats to be validated for a string field (§6.5.2).
Can be one of the following options:

- *NONE* - Default value. Means that the corresponding string field (§6.5.2) should not represent an ISBN.
- *ISBN10* - ISBN-10 code.
- *ISBN13* - ISBN-13 code.
- *ALL* - Both ISBN-10 and ISBN-13 codes.

ISSN style

Represents different ISSN formats to be validated for a string field (§6.5.2).
Can be one of the following options:

- *NONE* - Default value. Means that the corresponding string field (§6.5.2) should not represent an ISSN.
- *NORMAL* - Normal ISSN validation, allows lower case x at the end and non hyphenated ISSN values.
- *CASE_SENSITIVE* - Requires an upper case X at the end.
- *REQUIRE_HYPHEN* - Requires a hyphenated ISSN value.
- *STRICT* - Combines CASE_SENSITIVE and REQUIRE_HYPHEN.

Ip address scope

Represents different ways how to validate a given IP address.
Can be one of the following options:

- *NONE* - Default value. Means that the corresponding string field (§6.5.2) should not represent an IP address.
- *IP4* - Validates for IPv4 addresses in all ranges.
- *IP6* - Validates for IPv6 addresses in all ranges.
- *ALL* - Validates all IP formats in all ranges.
- *IP4_NO_PRIV* - Validates for IPv4 addresses without private ranges.
- *IP6_NO_PRIV* - Validates for IPv6 addresses without private ranges.
- *ALL_NO_PRIV* - Validates all IP formats without private ranges.
- *IP4_NO_RES* - Validates for IPv4 addresses without reserved ranges.
- *IP6_NO_RES* - Validates for IPv6 addresses without reserved ranges.
- *ALL_NO_RES* - Validates all IP formats without reserved ranges.
- *IP4_PUBLIC* - Validates for IPv4 addresses using only public ranges (without private and reserved ranges).
- *IP6_PUBLIC* - Validates for IPv6 addresses using only public ranges (without private and reserved ranges).
- *ALL_PUBLIC* - Validates all IP formats using only public ranges (without private and reserved ranges).

Text field

Represents a field type for storing larger text.
A text field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *length* - The length of this field. Default value is 2000.

In edit (§6.5.3) pages the generator will use multiline input elements (textarea).

User field

Extension of abstract integer field (§6.5.2) for storing user ids.

An user field has the following properties in addition to the common abstract integer field (§6.5.2) settings:

- *onAccountDeletion* - Controls how an app should change the field when users are deleted. Default value is *GUEST*. The available options are listed here (§6.5.2).

In edit (§6.5.3) pages the generator will implement an auto completion element allowing searching users by their name. For the output in view (§6.5.3) and display (§6.5.3) templates the user name is shown and linked to the corresponding user profile in case a profile module has been set in the Settings module administration.

Email field

Represents a field type for storing email addresses.

An email field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *checkMX* - A boolean specify whether the MX record's validity of the given email's host is checked or not. Only supported for the 1.4.x target core version.
- *checkHost* - A boolean specify whether the MX or A or AAAA record's validity of given email's host is checked or not. Only supported for the 1.4.x target core version.
- *length* - The length of this field. Default value is 255.

In edit (§6.5.3) pages the generator will use email input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) pages an icon will be shown linking the email address.

Url field

Represents a field type for storing urls.

An url field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *length* - The length of this field. Default value is 255.

In edit (§6.5.3) pages the generator will use url input elements as well as validation on client and server side. For the output in view (§6.5.3) and display (§6.5.3) pages an icon will be shown linking the url.

Upload field

Represents a field type for storing upload files.

An upload field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *allowedExtensions* - List of file extensions to be accepted during the upload, separated by a comma with a space char. Default value is *gif, jpeg, jpg, png*.
- *allowedFileSize* - Maximum file size in bytes, default is 0 for no limit.
- *length* - The length of this field. Default value is 255.
- *namingScheme* - Defines how uploaded files are named (§6.5.2).
- *subFolderName* - Name of sub folder for storing uploaded files. If this is empty the field name will be used as folder name.

In edit (§6.5.3) pages the generator will use upload input elements. If a field is mandatory the upload will be required when creating a new entity, but not when editing an existing one. If a field is optional (not mandatory) then it will be possible to delete existing uploads on editing.

For the output in view (§6.5.3) and display (§6.5.3) pages a download link is shown together with the filesize. If the file is an image then a small version of it is shown instead of a text link (on edit pages too by the way).

If an application has any upload fields the generator creates an additional util class containing methods for image processing. The generated application uses it to create and store thumbnails on demand with the help of the Imagine library which is included in Zikula 1.3. There is a view modifier available which works together with the mentioned util class and understands many parameters to use arbitrary images in the templates.

For every upload field *foo* there will be another array field (§6.5.2) created which is named *fooMeta*. This field stores some meta information about the uploaded files for convenience, like the file size, the image format (portrait, landscape, square) and the image dimensions.

Upload naming scheme

Represents different schemes for naming uploaded files.

Can be one of the following options:

- *ORIGINALWITHCOUNTER* - Keep the original file name. Add a counter if required to avoid duplicated file names.
- *RANDOMCHECKSUM* - Use a random checksum. This results in quite cryptic filenames.
- *FIELDNAMEWITHCOUNTER* - Use the field name as a prefix together with a counter. For example *image1*, *image2*, and so on.

Within the generated *upload handler* class one of those strategies will be selected depending on from which entity the currently treated upload file.

List field

Represents a field type for realising a selection of list values.

A list field has the following properties in addition to the common abstract string field (§6.5.2) settings:

- *length* - The length of this field. Default value is 255.
- *multiple* - A boolean specifying whether multiple items can be selected concurrently or not. The default value is *false*.
- *min* - Minimum amount of values enforced to be selected (only if *multiple* is set to *true*). The default value is *0*.
- *max* - Maximum amount of values enforced to be selected (only if *multiple* is set to *true*). The default value is *0*.
- *useChecks* - A boolean to enable a checkbox list (only if *multiple* is set to *true*). The default value is *false*.

A list field may have the following references:

- *items* - Allows referencing one or more items (§6.5.2).

The generator creates an additional class for handling the available list items centrally. Based on this information edit (§6.5.3) pages provide either a dropdown list (for single or multiple values depending on the *multiple* property) or a checkbox list (if *multiple* and *useChecks* are both set to *true*). For the output in view (§6.5.3) and display (§6.5.3) templates there is a modifier generated which cares for showing the names instead of the raw option values.

List field item

Represents an entry for a list field.

It includes the following properties:

- *default* - A boolean specifying whether this entry is selected by default or not. The default value is false.
- *name* - Name of the item.
- *image* - Optional name of an extrasmall image in the Zikula core, for example *xedit*.

See the list field (§6.5.2) section for a description of what the generator does with those elements.

Array field

Represents a field type for storing arrays.

An array field has the following properties in addition to the common entity field (§6.5.2) settings:

- *arrayType* - Which array type (§6.5.2) is used.
- *min* - Minimum amount of items enforced to be present. The default value is *0* which means that no certain amount is enforced. Must be used together with *max*.
- *max* - Maximum amount of items enforced to be present. The default value is *0* which means that no certain amount is enforced. Must be used together with *min*.

The generator will exclude arrays in edit (§6.5.3) pages as well as for the output in view (§6.5.3) and display (§6.5.3) templates.

Object field

Represents a field type for storing objects.

An object field has no fields or references in addition to the common entity field (§6.5.2) settings.

The generator will exclude objects in edit (§6.5.3) pages as well as for the output in view (§6.5.3) and display (§6.5.3) templates.

Abstract date field

Represents an abstract date dependant field for grouping those field types.

An abstract date field has the following properties in addition to the common entity field (§6.5.2) settings:

- *future* - A boolean specifying whether the value must be in the future or not.
- *past* - A boolean specifying whether the value must be in the past or not.
- *timestampable* - Which timestampable type (§6.5.2) is used.
- *timestampableChangeTriggerField* - Optional name of field to use as change trigger (if type is *CHANGE*. Can also be workflowState or the name of a relation (property.field).
- *timestampableChangeTriggerValue* - Optional value of field to use as change trigger (if type is *CHANGE*.

The *past* and *future* properties are implemented as client-side and server-side validators.

The generator transforms the timestampable attributes to the corresponding implementation as is. There are no differences made between the different timestampable types.

Datetime field

Represents a field type for storing datetime values with the format *YYYY-MM-DD H:i:s*.

A datetime field has the following properties in addition to the common abstract date field (§6.5.2) settings:

- *startDate* - A boolean specifying whether this field should be treated as a start date. If set to *true* this field is included into determining public visibility of the corresponding objects.
- *endDate* - A boolean specifying whether this field should be treated as an end date. If set to *true* this field is included into determining public visibility of the corresponding objects.
- *version* - A boolean specifying whether this field should act as a version. If set to *true* the owning entity will need to use optimistic locking (§6.5.2). Please read more at the integer field (§6.5.2) section. Also please note that it is preferred to use integer fields instead of datetime fields for version storage (read more in the validation chapter (§5.4.3)).

The generator will treat datetime values as date input elements with the *includeTime* attribute set to true in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *datetime* modifier is used to format the datetime according to the current locale.

Date field

Represents a field type for storing date values with the format *YYYY-MM-DD*.

A date field has the following properties in addition to the common abstract date field (§6.5.2) settings:

- *startDate* - A boolean specifying whether this field should be treated as a start date. If set to *true* this field is included into determining public visibility of the corresponding objects.
- *endDate* - A boolean specifying whether this field should be treated as an end date. If set to *true* this field is included into determining public visibility of the corresponding objects.

The generator will treat date values as date input elements with the *includeTime* attribute set to false in edit (§6.5.3) pages. For the output in view (§6.5.3) and display (§6.5.3) templates the *datetime* modifier is used to format the date according to the current locale.

Time field

Represents a field type for storing time values with the format *H:i:s*.

A time field has no fields or references in addition to the common abstract date field (§6.5.2) settings.

The generator will treat time values as text input elements with a maximum length in edit (§6.5.3) pages as long as there is no time field plugin implemented). For the output in view (§6.5.3) and display (§6.5.3) templates the *datetime* modifier is used to format the time according to the current locale.

Entity identifier strategy

Represents different strategies for identifier generation.

Can be one of the following options:

- NONE - No explicite strategy.
- AUTO - Choose automatically.
- SEQUENCE - Uses a database sequence.
- TABLE - Uses a single-row database table and a hi/lo algorithm.
- IDENTITY - Obtains IDs from special identity columns (auto_increment).
- UUID - Generates universally unique identifiers.
- CUSTOM - Custom strategy.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different index types. So beside the actual entity class there won't be any code parts affected based on which identifier strategy you use.

Entity change tracking policy

Represents different policies defining how changes are determined.

Can be one of the following options:

- *DEFERRED_IMPLICIT* - Compare properties during commit. Convenient, but not good for performance.
- *DEFERRED_EXPLICIT* - Scan only entities marked for change detection. Better performance, but no dirty checking.
- *NOTIFY* - Assume that entities inform listeners about their changes.

The generator transforms these values to the corresponding implementation as is. For *notify* the generator creates according notification calls within the entity setter methods.

Entity lock type

Represents different locking strategies for entities.
Can be one of the following options:

- NONE - No locking support.
- OPTIMISTIC - Optimistic locking.
- PESSIMISTIC_READ - Pessimistic read locking.
- PESSIMISTIC_WRITE - Pessimistic write locking.
- PAGELOCK - Use PageLock module.
- PAGELOCK_OPTIMISTIC - Use PageLock module combined with optimistic locking.
- PAGELOCK_PESSIMISTIC_READ - Use PageLock module combined with pessimistic read locking.
- PAGELOCK_PESSIMISTIC_WRITE - Use PageLock module combined with pessimistic write locking.

The generator transforms these values to the corresponding implementation as is. If you use optimistic locking the entity needs a version field which can be an integer (§6.5.2) or datetime (§6.5.2) field whereby an integer is preferred.

If you choose an option including the PageLock module the form handlers generated for the edit (§6.5.3) pages will call some api functions of the PageLock extension in order to incorporate these features.

Entity tree type

Represents different tree strategies for entities.
Can be one of the following options:

- NONE - No tree.
- NESTED - Nested set.
- CLOSURE - Closure.

If an entity has a tree type other than *NONE* then the generator creates several additional artifacts, like for example:

- An additional template for managing the tree in a hierarchy view.
- An additional view plugin for including the Zikula tree javascript.
- An additional template included in display pages for showing different types of relatives.
- Some ajax functions used by the hierarchy view.
- For closure: separate classes for the closure entities.

Entity slug style

Represents different slug styles for the creation of permalinks.
Can be one of the following options:

- LOWERCASE - Lowercase.
- UPPERCASE - Uppercase.
- CAMEL - Camelcase.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different slug styles. So beside the actual entity class there won't be any code parts affected based on which slug style you use.

Array type

Represents different types of arrays used by Doctrine 2.
Can be one of the following options:

- ARRAY - A normal array.
- SIMPLE_ARRAY - A simple array represented by a comma-separated text field.
- JSON_ARRAY - A JSON array.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different array field types. So beside the actual entity class there won't be any code parts affected based on which array type you use.

Entity timestampable type

Represents different events for triggering the Timestampable extension.
Can be one of the following options:

- NONE - No Timestampable extension.
- UPDATE - On update.
- CREATE - On create.
- CHANGE - On property change.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different timestampable types. So beside the actual entity class there won't be any code parts affected based on which timestampable type you use.

Entity workflow type

Represents different workflows for entities.
Can be one of the following options:

- NONE - No approval.
- STANDARD - Single approval.
- ENTERPRISE - Double approval.

In total there are nine different workflow states which are explained below. Note that you can arrange many states by corresponding properties in the model.

1. Initial - pseudo-state for content which is just created and not persisted yet.
2. Deferred - content has not been submitted yet or has been waiting, but was rejected. Only available if the entity has set the *ownerPermission* property to *true*. Allows users to manage their contributions. Otherwise rejected content would be deleted.
3. Waiting - content has been submitted and waits for approval. Only available for *STANDARD* and *ENTERPRISE*. Fetched for pending content integration and moderation panel.
4. Accepted - content has been submitted and accepted, but still waits for approval. Only available for *ENTERPRISE*. Fetched for pending content integration and moderation panel.
5. Approved - content has been approved and is available online.
6. Suspended - content has been approved, but is temporarily offline. Only available if the entity has set the *hasTray* property to *true*.
7. Archived - content has reached the end and became archived. Only available if the entity has set the *hasArchive* property to *true*. Requires a datetime (§6.5.2) or date (§6.5.2) field being designated as end date.
8. Trashed - content has been marked as deleted, but is still persisted in the database. Only available if the entity has set the *softDeleteable* property to *true*. Note that the soft deleteable implementation is not done yet as this requires the 1.4.0 core of Zikula.
9. Deleted - pseudo-state for content which has been deleted from the database.

The following image shows an overview of all possible workflow states and actions.
The current state for a certain object is stored in the workflow itself. Furthermore ModuleStudio creates an additional field named *workflowState* to each entity before starting the generation. This allows for easier filtering and other useful applications without having to load the workflow in all cases. In fact ModuleStudio adds it as a list field (§6.5.2) which contains a list field item (§6.5.2) for each state.

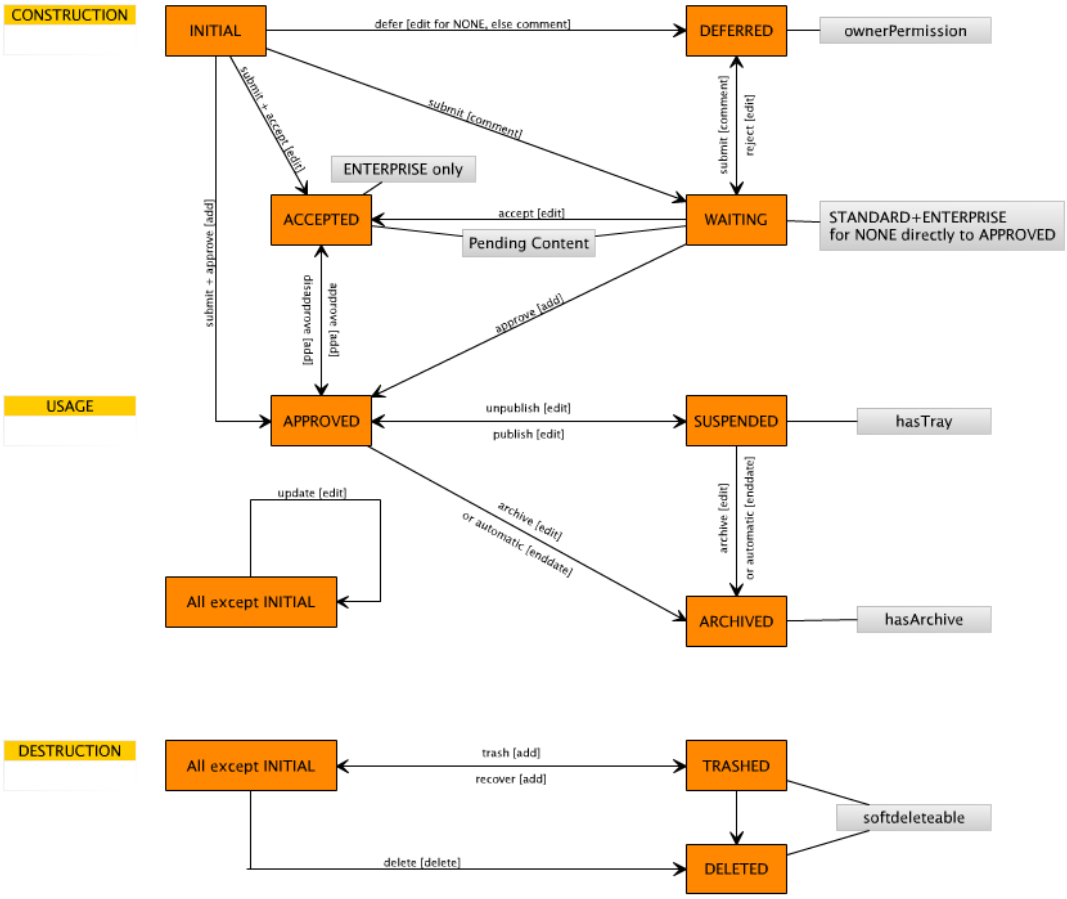

Figure 6.4.: Workflow overview

Note that it is easily possible to model date or datetime fields (§6.5.2) which set their value automatically depending on a certain workflow state. Just set their timestampable type to *CHANGE* and set *workflowState* as change trigger field. Also set the change trigger value to the name of the desired state. So you could for example create an approval date by using *approved* for the trigger value property.

For all entities having another workflow than *NONE* there are configuration options in the generated configuration page for selecting user groups for moderation. If these settings are not applied, the default group for administrators is used as fallback. With the help of this information, email notifications are sent between creator and moderators on state changes. For moderators there is a textarea field provided in the form for specifying additional remarks, like a reason for deny (particularly useful for *reject* and *demote* actions).

Account deletion handler

Represents the kind of reaction to perform if users are deleted, but still referenced in some data of the generated application.

Can be one of the following options:

- ADMIN - Sets the corresponding user id to 2 (admin user).
- GUEST - Sets the corresponding user id to 1 (guest).
- DELETE - Deletes the entity.

Entity index

Represents an entity index.

It includes the following properties:

- *type* - The index type.

An index may have the following references:

- *entity* - Reference to the owning element.
- *items* - Allows referencing one or more index items (§6.5.2).

Entity index type

Represents different types of entity indexes (§6.5.2).

Can be one of the following options:

- NORMAL - Normal index.
- UNIQUE - Unique constraint.

The generator transforms these values to the corresponding implementation as is. There are no differences made between the different index types. So beside the actual entity class there won't be any code parts affected based on which index type you use.

Entity index item

Represents a part of an index (§6.5.2), referencing to an equally-named entity field (§6.5.2).

An index item may have the following references:

- *index* - Reference to the owning element.

Relationship

Base class for all types of associations between entities.
It includes the following properties:

- *bidirectional* - A boolean specifying whether this relationship is bidirectional or not. The default value is *false* for performance reasons.

A relationship may have the following references:

- *container* - Reference to the owning element.
- *source* - Allows referencing a target entity (§6.5.2).
- *target* - Allows referencing a source entity (§6.5.2).

Join relationship

Collects all foreign key and join relationships.
It includes the following properties in addition to the common relationship (§6.5.2) settings:

- *cascade* - The cascade type (§6.5.2) used on application level from source view.
- *cascadeReverse* - The cascade type (§6.5.2) used on application level from target view (only for bidirectional relationships).
- *editType* - The edit type (§6.5.2) for this association, only applicable if *useAutoCompletion* is not set to *NONE*.
- *fetchType* - The fetch type (§6.5.2) for this association.
- *nullable* - A boolean specifying whether the field for this relationship may be null or not. The default value is *true*.
- *onDelete* - String for optional update cascade options on database level (for example *RESTRICT* or *SETNULL*).
- *onDelete* - String for optional delete cascade options on database level (for example *RESTRICT* or *SETNULL*).
- *useAutoCompletion* - If set to any value except *NONE* the generator will create an auto completion field instead of a normal dropdown select field for the corresponding side(s) of the relationship. For more information see the available options (§6.5.2).
- *sourceAlias* - The alias for the source entity, required to have multiple associations between the same entities. The name should reflect the cardinality on the source side (singular or plural forms) depending on the relationship type. As with all names camel case is preferred, for example *personAddresses*.

- *sourceField* - Name of the source entity fields used for the join. The default value is *id* which means that the source entity is joined by it's primary key. It is possible to change that value for custom join conditions. Furthermore it is possible to use multiple field names separated by a comma with a space in order to join entities with composite keys.
- *targetAlias* - The alias for the target entity, required to have multiple associations between the same entities. The name should reflect the cardinality on the target side (singular or plural forms) depending on the relationship type. As with all names camel case is preferred, for example *personAddresses*.
- *targetField* - Name of the target entity fields used for the join. The default value is *id* which means that the target entity is joined by it's primary key. It is possible to change that value for custom join conditions. Furthermore it is possible to use multiple field names separated by a comma with a space in order to join entities with composite keys.
- *unique* - A boolean specifying whether the field for this relationship is unique or not. The default value is *false*.

The generator transforms most of these settings to the corresponding implementation as is. The only thing which is used outside of the entity classes is the edit type (§6.5.2) which controls how relationships are handled in edit actions (§6.5.3).

Join relationships are automatically incorporated into the dql queries which are placed in the entity repository classes. You can override these methods for changing selection details if required.

One to one relationship

Represents one-to-one relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *orphanRemoval* - Default value is *false*. If set to *true* orphans get removed automatically.
- *primaryKey* - A boolean specifying whether the foreign key of this relation should act as a primary key. The default value is *false*. Please note that this has not been tested yet and probably won't be supported properly yet by the controller layers in the generated application.

One to many relationship

Represents one-to-many relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *indexBy* - Set to target field name (must be unique) to specify the index by criteria for the relation. Please note that this has not be tested very well yet.
- *orderBy* - Set to target field name to specify the sorting criteria for the outgoing relation.
- *orphanRemoval* - Default value is *false*. If set to *true* orphans get removed automatically.
- *minTarget* - Minimum amount of items enforced to be present on the target side. The default value is *0* which means that no certain amount is enforced. Must be used together with *maxTarget*.
- *maxTarget* - Maximum amount of items enforced to be present on the target side. The default value is *0* which means that no certain amount is enforced. Must be used together with *minTarget*.

Many to one relationship

Represents many-to-one relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *primaryKey* - A boolean specifying whether the foreign key of this relation should act as a primary key. The default value is *false*. Please note that this has not been tested yet and probably won't be supported properly yet by the controller layers in the generated application.

Many to many relationship

Represents many-to-many relationships.

It includes the following properties in addition to the common join relationship (§6.5.2) settings:

- *indexBy* - Set to target field name (must be unique) to specify the index by criteria for the relation. Please note that this has not be tested very well yet.
- *orderBy* - Set to target field name to specify the sorting criteria for the outgoing relation.
- *orderByReverse* - Set to source field name to specify the sorting criteria for the incoming relation.
- *refClass* - Specifies the reference class created for the linking table (for example *personAddress*). The generator creates additional classes for this reference-managing entity.
- *minSource* - Minimum amount of items enforced to be present on the source side. The default value is *0* which means that no certain amount is enforced. Must be used together with *maxSource*.

- *maxSource* - Maximum amount of items enforced to be present on the source side. The default value is *0* which means that no certain amount is enforced. Must be used together with *minSource*.
- *minTarget* - Minimum amount of items enforced to be present on the target side. The default value is *0* which means that no certain amount is enforced. Must be used together with *maxTarget*.
- *maxTarget* - Maximum amount of items enforced to be present on the target side. The default value is *0* which means that no certain amount is enforced. Must be used together with *minTarget*.

Cascade type

Represents different cascade types on application level.
Can be one of the following options:

- NONE
- PERSIST
- REMOVE
- MERGE
- DETACH
- PERSIST_REMOVE
- PERSIST_MERGE
- PERSIST_DETACH
- REMOVE_MERGE
- REMOVE_DETACH
- MERGE_DETACH
- PERSIST_REMOVE_MERGE
- PERSIST_REMOVE_DETACH
- PERSIST_MERGE_DETACH
- ALL

The cascade type is implemented as defined in the association's annotation within the corresponding entity classes. At the moment there are no other code parts depending on that.

Auto completion usage

Defines whether and which sides of a relationship will be handled by an auto completion field instead of a dropdown field during editing. Note that inline creation and editing of related items (see edit types (§6.5.2)) is only possible when using the auto completion approach.

Can be one of the following options:

- NONE - Use no auto completion.
- ONLY_SOURCE_SIDE - Use auto completion when selecting entities of the source side (editing of the target side).
- ONLY_TARGET_SIDE - Use auto completion when selecting entities of the target side (editing of the source side).
- BOTH_SIDES - Use auto completion on both sides.

Actually these settings are part of the view layer (§6.5.4) and actually they are going to be moved somewhen in future. But this is a case where we decided that we don't want to wait until the view layer is ready to be used (as priorities require to do other things first).

Example for inline editing:

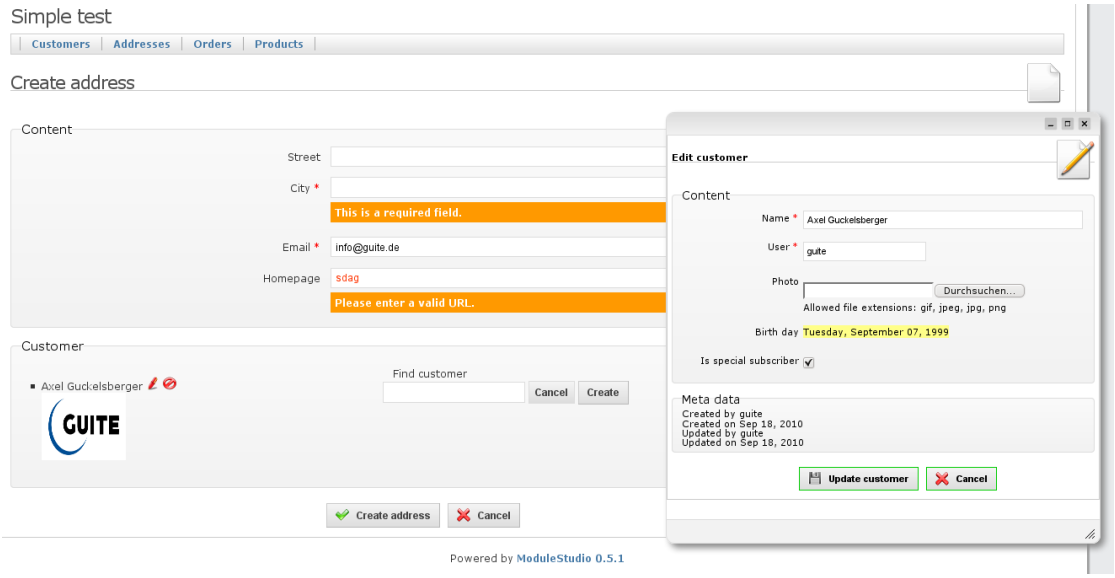

Figure 6.5.: Inline editing

Relation fetch type

Represents different fetch types for join relationships.

Can be one of the following options:

- LAZY - Lazy.
- EAGER - Eager.
- EXTRA_LAZY - Extra lazy.

The generator transforms these values to the corresponding implementation. There are no differences made yet between the different fetch types as the generator uses DQL for almost all selections. So beside the actual entity class there won't be any code parts affected based on which fetch type you use.

Relation edit type

Represents different edit types for join relationships.

Can be one of the following options:

- ACTIVE_NONE_PASSIVE_CHOOSE - Editing the parent does nothing. Editing the child includes choosing the parent.
- ACTIVE_NONE_PASSIVE_EDIT - Editing the parent does nothing. Editing the child includes choosing, adding and editing the parent.
- ACTIVE_CHOOSE_PASSIVE_NONE - Only for many-to-many: Editing the parent includes choosing the children. Editing the child does nothing.
- ACTIVE_EDIT_PASSIVE_CHOOSE - Editing the parent includes choosing, adding and editing the children. Editing the child includes choosing the parent.
- ACTIVE_EDIT_PASSIVE_EDIT - Editing the parent includes choosing, adding and editing the children. Editing the child includes choosing, adding and editing the parent.
- ACTIVE_EDIT_PASSIVE_NONE - Only for many-to-many: Editing the parent includes choosing, adding and editing the children. Editing the child does nothing.

Actually these settings are part of the view layer (§6.5.4) and actually they are going to be moved somewhen in future. But this is a case where we decided that we don't want to wait until the view layer is ready to be used (as priorities require to do other things first).

For each entity the generator creates some templates to be included in the edit templates of related entities (for example a display list and another one for edit). Depending on which edit type is defined for a relationship the corresponding edit template (choose or edit) is included or not.

- NONE means that there is no possibility to take influence on the association.
- CHOOSE means that it is possible to select a related entity with the help of auto completion.
- EDIT means the same as CHOOSE plus that it is also possible to created and edit related entities during editing the main entity.

Inheritance relationship

Represents inheritance relationships for describing entity class hierarchies.

It includes the following properties in addition to the common relationship (§6.5.2) settings:

- *discriminatorColumn* - Name of the field used for storing the entity type.
- *strategy* - The inheritance strategy used for data storage.

The generator considers inheritance for all classes which are created for each entity. This includes naturally the entity classes itself, but also additional classes like repositories, validators or additional entities for extensions like attributes, categories, log entries, translations and more.

As explained in the entity section (§6.5.2) all generated concrete classes inherit from corresponding abstract base classes. As soon as an entity does inherit from another one, there will be no base class created for it. Instead the concrete implementation class will inherit from the concrete class of the parent entity.

Example for Zikula 1.4.0:

```
namespace MyModule\Entity\Validator\Base;
 class Person extends \MyModule\Validator
{
    // generator code
}
 namespace MyModule\Entity\Validator;
 class Person extends Base\Person
{
    // manual code
}
 namespace MyModule\Entity\Validator;
 class Customer extends Person
{
    // manual code
}
```

Example for Zikula 1.3.7 and below:

```
class MyModule_Entity_Validator_Base_Person
    extends MyModule_Validator
{
    // generator code
}
 class MyModule_Entity_Validator_Person
    extends MyModule_Entity_Validator_Base_Person
```

```
{
    // manual code
}
 class MyModule_Entity_Validator_Customer
    extends MyModule_Entity_Validator_Person
{
    // manual code
}
```

While this implementation approach is quite elegant it is not completed yet in all areas unfortunately. At least during installation everything should be fine. When working with the application you will notice that inherited fields are handled well, but additional fields from the parent classes are not considered yet everywhere. See #46 for more information.

Inheritance strategy type

The strategy type defines which kind of inheritance strategy should be used.
Can be one of the following options:

- SINGLE_TABLE - Simple inheritance: share everything and store it in the parent table.
- JOINED - Concrete inheritance: each entity stores everything in its own table.

The generator transforms these values to the corresponding implementation. There are no differences made yet between the different strategies. So beside the actual entity class there won't be any code parts affected based on which strategy you use.

Variables

Container class for carrying module vars.
It includes the following properties:

- *sortOrder* - The sorting position for when using multiple variable sections.

A var container may have the following references:

- *container* - Reference to the owning element.
- *vars* - Allows referencing one or more variables (§6.5.2).

As soon as at least one variable container exists the generator creates a *config* page in the admin area to let the site admin manage corresponding settings.
If a model contains multiple variable containers the config page will use a tabbed panel containing a tab for each container, sorted by the sortOrder field. This allows separating settings in bigger models into logical semantic groups.

Variable

Represents a module variable.
It includes the following properties:

- *name* - Name of the variable.
- *value* - Default value of the variable.

A variable may have the following references:

- *container* - Reference to the owning element.

For each variable the generator creates an according input element in the config page. Also the variable is handled properly in the installer classes which takes care for initialisation and removal on uninstallation.
If you enabled interactive installation there will also be an init page asking for the initial values for all variables. However this is not matured very well yet.

Text var

Represents a setting with alphanumeric values.
It includes the following properties in addition to the common variable (§6.5.2) settings:

- *maxLength* - The maximum length.

The generator creates an input for text for a text variable. The maximum length is not considered anywhere in the generated code yet.

Int var

Represents a setting with numeric (integer) values.
The generator creates an input element for integers (digits) for an integer variable.

Bool var

Represents a setting with boolean values.
The generator creates a checkbox input element for a boolean variable.

File path var

Represents a setting with file path values.
It includes the following properties in addition to the common variable (§6.5.2) settings:

- *withinDocRoot* - A boolean specifying whether this path is placed inside the web root or not. The default value is *true*.

- *writable* - A boolean specifying whether this file path is writable or not. The default value is false.

The generator will create a text input element for a file path variable as well as additional behaviour to consider it's fields properly. At the moment this has not been done yet though, so file path vars are currently handled in the same way like text vars.

List var

Represents a setting with list values.

It includes the following properties in addition to the common variable (§6.5.2) settings:

- *multiple* - A boolean specifying whether multiple items can be selected concurrently or not. The default value is *false*.

A list variable may have the following references:

- *items* - Allows referencing one or more items (§6.5.2).

The generator will create a select element for a list variable. At the moment this has not been done yet though, so list vars are currently handled in the same way like text vars.

List var item

Represents an entry for a setting with list values.

It includes the following properties:

- *default* - A boolean specifying whether this entry is selected by default or not. The default value is false.
- *name* - Name of the item.

The generator will create an option element for the corresponding select element. At the moment this has not been done yet though, so list vars are currently handled in the same way like text vars.

Entity event listener

Base class for model elements representing entity event listeners.

List of supported events:

- PrePersist / PostPersist
- PreUpdate / PostUpdate
- PreRemove / PostRemove

- PostLoad

An event listener may have the following references:

- *operations* - Allows referencing one or more transform objects (§6.5.2).

Event listeners are not part of the model editor yet and are therefore ignored by the generator at the moment. Instead the generator creates simply all listener methods for all entities.

Transform object

Base class for transformation objects encapsulating algorithm puzzle pieces.
Examples:

- Arithmetic operations
- Datetime operations
- Financial operations
- Logical operations
- String operations, e.g. replacements
- System calls

A transform object may have the following references:

- *container* - Reference to the owning element.

Transform objects are not part of the model editor yet and are therefore ignored by the generator at the moment.

Combinations and edge cases

This section is going to collect certain combinations of elements in practical scenarios. The intention of this section is to point out dependencies and other essential aspects relating the combination of model elements in various ways.

6.5.3. Controller layer

Language elements

Controller

A controller represents an area with functions which are called actions (§6.5.3). In Zikula a controller is identified with the *type* parameter.
The following controller types are available:

- Admin controller - for admin areas.
- User controller - for user areas.
- Ajax controller - for ajax functions.
- Custom controller - for custom areas, like *edit*.

A controller may have the following references:

- *actions* - Allows referencing one or more actions (§6.5.3).
- *container* - Reference to the owning element.
- *handlers* - Allows referencing one or more action handlers (§6.5.3).

The generator creates controller classes with methods for each actions. Action handlers are not considered yet.

If you have added some variables (§6.5.2) the admin controller will contain a *config* method for managing the modvar settings.

Action

An action represents a controller function which can be called by the user. In Zikula an action is identified with the *func* parameter.

The following action types are available:

- Main action - default function.
- View action - processes a collection of entities.
- Display action - shows a certain entity in detail.
- Edit action - an action for editing an entity.
- Delete action - an action for deleting an entity.
- Custom action - for custom actions, like *mySpecialFunction*.

An action may have the following references:

- *controller* - Reference to the owning element.
- *handler* - Allows referencing an action handler (§6.5.3).
- *incoming* - Allows referencing one or more transitions (§6.5.3).
- *outgoing* - Allows referencing one or more transitions (§6.5.3).

The generator creates sensitive default implementations for all action types except custom actions which do only return an empty template.

It is possible to create special versions for all templates by adding a suffix which will be assigned by the *tpl* parameter. For example you can call *display_myversion.tpl* by adding *&tpl=myversion* to the url. Note this additional override capability is mainly intended for all actions which have a template for each entity and has therefore not been considered for main and custom actions yet.

Main action

A main action implementation does either do a simple redirect to the view function or (if no view is available) fetch a simple template file.

View action

The view implementation offers a generic list view of multiple items which can be sorted and filtered. Also there are alternative template formats created for atom, csv, json, rss, xml and maybe kml support. Just add *&userssext=1* to the url or, even easier with shorturls, change *persons.html* to *persons.rss*.

Display action

A display action results in a generic detail view of an entity. Again there are alternative formats supported, like for example csv, json and xml.

Edit action

The edit implementation creates form pages for changing entities and their relations. Beside the form handler classes this involves according template files.

Delete action

For a delete action the generator creates the well-known confirmation page asking the user whether he really wants to delete the given entity.

Custom action

A custom action is only created as a mockup which contains the permission check as well as some other stuff which is always required, like returning the output from a template fetched by the view.

Action handler

Action handlers encapsulate the reaction on user interactions. The most common instance is a form handler processing and validating some input fields.

The following handler types are available:

- List handler - reacts on interactions with a view action (§6.5.3) (e.g. for filtering and sorting).
- Detail handler - reacts on interactions with a display action (§6.5.3).
- Edit handler - reacts on interactions with an edit action (§6.5.3).
- Custom handler - reacts on interactions with a custom action (§6.5.3).

A handler may have the following references:

- *events* - Allows referencing one or more events (§6.5.3).
- *action* - Reference to the owning element.
- *container* - Reference to the owning element.
- *controller* - Allows referencing a controller (§6.5.3).

Not used yet by the generator at all.

Action event

An action event represents something which can happen within the scope of an action handler. For example this could be the press of a button.

The following event types are available:

- Start / initial event.
- Normal event.
- End / final event.

An event may have the following references:

- *handler* - Reference to the owning element.

Not used yet by the generator at all.

Transition

Transitions define how the user may move between the available use cases defined by the controller actions. (§6.5.3)

The following transition types are available:

- Redirect - automatic.
- User transition - manual.

It includes the following properties:

- *condition* - A conditional expression which must be or become true to activate the transition.

A transition may have the following references:

- *container* - Reference to the owning element.
- *source* - Reference to source action.
- *sourceEvent* - Reference to the source action's event causing this transition.
- *target* - Reference to target action.

Not used yet by the generator at all.

Combinations and edge cases

At the moment the controller editor is almost limited to the creation of controller and action elements. As stated above action handlers and action events are not really used yet. Therefore there the variation in this layer is not that huge yet, so there are no special aspects to explain here.

6.5.4. View layer

As there is no view editor available yet all the following elements are not relevant for the generator yet. Instead it only creates default templates foreach existing controller action (§6.5.3) and entity (§6.5.2), as well as some common templates which are included or required for some extensions.

Language elements

View

The base view class. Each view instance would correspond to a template in the generated Zikula application.

The following view types are available:

- List view.
- Detail view.
- Edit view.
- Custom view.

A view may have the following references:

- *viewContainer* - Reference to the owning element.
- *viewRoot* - Reference to root panel.

Tables

The following table types are available:

- Layout table (is a composite container (§6.5.4)).
- Data table (is a sub container (§6.5.4)).

Container

Represents a very generic container.
A container may have the following references:

- *cancelActvators* - Allows referencing one or more cancel activators (§6.5.4).
- *submitActivators* - Allows referencing one or more submit activators (§6.5.4).

Root panel

Top-level container for a certain view.
A root panel may have the following references in addition to the generic container fields:

- *containers* - Allows referencing one or more composite containers (§6.5.4).
- *presenter* - Reference to the owning element.

Composite container

A composite container which may include other sub containers.
A composite container may have the following references in addition to the generic container fields:

- *childContainer* - Allows referencing one or more sub containers (§6.5.4).
- *rootPanel* - Reference to the owning element.

Tabbed pane

A composite container which groups its sub containers with tabs.

Sub container

A sub container which may not include other sub containers.
A sub container may have the following references in addition to the generic container fields:

- *fields* - Allows referencing one or more fields (§6.5.4).
- *parentContainer* - Reference to the owning element.

Panel

Inherits from both composite container (§6.5.4) and sub container (§6.5.4).

Search panel

A dedicated panel for search functions.

A search panel may have the following references in addition to the sub container fields:

- *searchButton* - Allows referencing a submit button (§6.5.4).

Form

Extension of sub container (§6.5.4) for form elements.

Field

Represents a field with information.

It includes the following properties:

- *cssClass* - Optional specification of arbitrary css classes.
- *defaultValue* - The default value (which may be different from that in the data layer).

A field may have the following references:

- *fieldContainer* - Reference to the owning element.

Display field

Shows the value of a certain field from the data layer.

Input field

Shows an input for a certain field from the data layer.

It includes the following properties in addition to the common field (§6.5.4) settings:

- *hasInitialFocus* - Whether this input should receive the initial focus or not.
- *id* - Defines the markup element id.
- *isReadOnly* - Whether this input is read only or not. *Must be revalidated later to see if this is really required.*
- *isMandatory* - Whether this input is mandatory or not (might be different as specified in the data layer).
- *isVisible* - Whether this input is visible or not.

Design field

Shows some additional information which is not data-driven.
It includes the following properties in addition to the common field (§6.5.4) settings:

- *tagName* - Defines which markup tag should be used (for example *p* or *h4*.

Form label

Design field (§6.5.4) extension for form labels.
It includes the following properties in addition to the design field (§6.5.4) settings:

- *for* - Allows referencing an input field (§6.5.4).

Button

Abstract representation for buttons.
A button may have the following references:

- *label* - String for the label text.

Link

Abstract representation for hyperlinks.

Activator

An activator is something which submits a form or start another way of interaction.

Submit button

Inherits from both submit activator (§6.5.4) and button (§6.5.4).
A submit button may have the following references:

- *searchContext* - Allows referencing a search panel (§6.5.4).

The submit button type defines which kind of cancel button should be used:

- OK
- YES
- CONTINUE
- NEXT

Cancel button

Inherits from both cancel activator (§6.5.4) and button (§6.5.4).
The cancel button type defines which kind of cancel button should be used:

- CANCEL
- NO
- PREVIOUS

Submit activator

A submit activator may have the following references:

- *submitContext* - Allows referencing a container (§6.5.4).

Cancel activator

A cancel activator may have the following references:

- *cancelContext* - Allows referencing a container (§6.5.4).

Submit link

Inherits from both submit activator (§6.5.4) and link (§6.5.4).

Back link

Inherits from both cancel activator (§6.5.4) and link (§6.5.4).

Layout order

With layout orders you can add relations to panels to define whether they should float beside each other or not.
It includes the following properties:

- *length* - The length value.
- *lengthtype* - The length type (see below). Default is PIXELS.
- *type* - The type of layout order (see below). Default is HORIZ_LEFT.

A layout order may have the following references:

- *source* - Reference to source sub container.
- *target* - Reference to target sub container.
- *viewContext* - Reference to the owning element.

Layout order type

Defines how the panels are connected to each other visually.
Can be one of the following options:

- HORIZ_LEFT
- HORIZ_RIGHT
- VERTICAL

Layout order length type

Specifies the unit which is used for the length definition.
Can be one of the following options:

- PIXELS
- PERCENT
- EM
- EX
- CENTIMETER
- MILLIMETER
- INCH
- POINTS
- PICAS

Combinations and edge cases

As there is no view editor available yet this section is empty for now.

6.5.5. Workflow layer

The workflow layer is not very configurable yet. At the moment there are three predefined workflows available which can be defined in the model for each entity (§6.5.2). For more information see entity workflow types (§6.5.2).

In future it is planned to create a dedicated layer for modeling only possible workflow states and actions.

6.6. Additional notes

None yet.

7. Customisation and maintenance

7.1. Introduction

This section gives a few hints for changing arbitrary things in your generated applications while keeping in sync with upcoming releases of Zikula and ModuleStudio.

7.2. Long-term maintenance

To prevent losing the benefits of ModuleStudio (§1.2) in future you have to follow a few simple basic rules in your development process.

7.2.1. Keep consistent

When using ModuleStudio then your model is not only some sort of bootstrapping or documentation artifact. The model describes or better is the real application. Therefore you should do all important changes (like adding or moving table columns, renaming an entity or other amendments, introducing a new controller action, etc.) on model level although it might look a bit inconvenient first. Do not let your model become obsolete, as this would mean losing lots of advantages. You will thank yourself when generating again with a newer version.

7.2.2. Document your changes

Document your changes to simplify the merging process you will have to do after regeneration. For example after you added some fields later on, or you got a new generator version fixing some bugs, and so on you will want to know again where you did which changes for which reason. Usually a good place for this documentation is *modules/YourModule/docs/customisation.txt*.

7.2.3. Use overriding

All cosmetic enhancements can be done by template overriding in Zikula. For example you can place custom templates in */config/templates/YourMod/...* during development. *Note* that if legacy mode is turned off, template overriding will not work automatically. In the past Zikula performed a file system scan several directories for each template which was very convenient, but also bad for performance. Now you have to describe overrides in the file *template_overrides.yml* explicitely instead.

So when custom templates are placed in the */config/templates/YourMod/...* folder this is fine for development. It is even recommended for the module developer to keep things separated during development (and in Git, see below) so that he can always distinguish

(and compare) between generated and customised templates, since this makes things easier if the module is regenerated again in future.

When using a module in production things are a bit different. A site owner may want to further customise some templates, so the */config/* directory should not be used by the module itself out of the box. This is not really a technical problem, but just a question of how the deployment process should be. You can for example move the overridden templates into the module before a release.

If you need additional display-oriented logic, simply create a view plugin encapsulating your efforts in a file which is not affected by the generator at all.

7.2.4. Code additions

Perform logical enhancements in the generated implementation classes. These extend from abstract base classes containing the actual generator implementation code. So almost all concrete classes are empty waiting for your custom extension.

The *ControllerUtil* class can be used to enable/disable specific controller actions (like view, display, ...) and additional functionality (like blocks, mailz api, content type, etc.) for particular entity types within custom conditions.

When using edit forms you can easily customise the redirect behaviour by appending a *returnTo* parameter to the edit url. For example let's imagine you modeled a customer entity having a bidirectional relationship to many addresses. In this case the default behaviour of the *edit address* functionality redirects back to the detail view of the created/updated entity if it exists in the model. If there is no display action it redirects to the list view if existing else it jumps to the index page. Using the *returnTo* parameter you can assign certain pages like *display*, *view* and even related items like *customerDisplay*.

7.2.5. Use versioning

Using a version control system, like git or svn, gives you another additional level of rollback safety and is a good idea anyway.

7.3. Additional notes

Please see also this tutorial. Some parts are a little outdated due to the migration from Doctrine 1.3 to Doctrine 2, but this shouldn't be a problem for the understandability of the overall messages shown there.

8. Other cartridges

8.1. Introduction

Beside an application it is possible to create other interesting artifacts from a given input model. This section explains what the other generator cartridges of ModuleStudio are doing.

8.2. Reporting

The *reporting* cartridge aims on generating helpful documents for the project management and application design. The following sections summarise the existing reports which are not feature complete, but act as a proof of concept for the integration of reporting technology.

8.2.1. Documentation

This document is a draft for a structural documentation of the model. It shows all essential model elements together with their most important properties. In case you added information to the generic *documentation* field of your model elements those descriptions will be shown here, too.

8.2.2. Model information

General document (used as a playground to experiment with things). Beside some statistics it shows some diagrams illustrating some measures with regards to the entities in the data layer. For example one can see which entities have the most relationships (and thus complexity to be managed).

8.2.3. Function points

This report shows a calculation of (unadjusted) function points (wikipedia). These are a measurement for the amount of complexity in a software system. It can be used as an input value for other methods like COCOMO II (wikipedia).

8.3. Additional notes

None yet.

9. AddOns

9.1. Introduction

This chapter is about extending ModuleStudio.

There will be different types of extensions for ModuleStudio, but this page is primarily intended for future versions.

9.1.1. Language packs

At the moment only German locales are shipped with ModuleStudio. These involve all own translation strings, but not those from the Eclipse platform and used frameworks yet. It is planned to make language packs available as optional feature bundles at a later stage.

9.1.2. Figure galleries

Not implemented yet though as the modeling language itself must become matured first.

9.1.3. Template sets

Not implemented yet though as the modeling language itself must become matured first.

9.1.4. Generator cartridges

Not implemented yet though as the modeling language itself must become matured first.

9.1.5. Other extensions

As the roadmap is quite open in the long term many aspects are not decided yet. Therefore things may change significantly.

9.2. Extension Points

None yet.

9.3. Additional notes

None yet.

Part III.

Appendix

10. Troubleshooting

10.1. Introduction

This sections collects information snippets which may be helpful when running into problems.

10.2. General hints

10.2.1. Pathes with space chars

Eclipse has problems with pathes containing space chars. Therefore avoid spaces to ensure things work correctly.

10.3. Tooling problems

10.3.1. Boolean properties can't be unset

Due to a bug it can happen that unsetting a boolean flag will not be persisted in the model file. To work around this please remove the flag using a text editor.

10.3.2. I can't open my model again

When reopening a model does not work, but just an empty editor tab is shown, something in the workspace data became stale or corrupt. To fix this please delete the folder named *MostWorkspace* in your user directory.

See also ticket 254 for some information about that problem.

10.3.3. The diagram looks broken

If model and diagram do not fit together anymore (which can happen when the model language evolved) you see something like the following.

This is no big problem though as you can create a new diagram for your application model with ease:

1. Close the model
2. Delete the .mostdiagram file
3. File -> Initialize diagram file

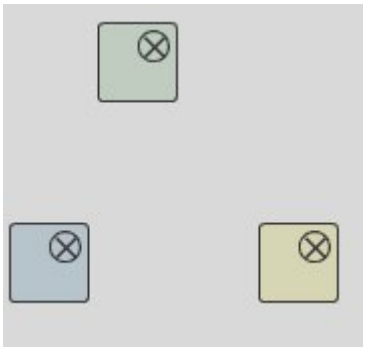

Figure 10.1.: Broken diagram

4. Choose the .mostapp file
5. Click on the next button
6. Click on the finish button

10.3.4. I can't save my model

If you get an error message when trying to save your model this means that it is not possible to serialise the object graph you have currently opened in the memory. The probable reason for this is that there exists a reference to an element without a name. Therefore the serialiser sees no way to persist this reference.

For example if you have two entities and a relationship between them then all these three elements need a name. Because if an entity has no name then the relationship can not store it's source or target reference. If the relationship itself has no name then the entities can not store it in their list of incoming or outgoing relationships.

To fix this just ensure that all existing elements have a name. In a later version the ModuleStudio tooling is going to get more support for setting sensitive default values to overcome this issue, but this feature addition has no high priority at the moment.

10.4. Migration problems

10.4.1. Migrating my existing model causes an error

Before converting your existing model make sure that you have removed any account controller and search controller elements in MOST 0.5.4.

10.5. Generation problems

10.5.1. Unable to find workflow file

If you get this error you have generated an application for Zikula 1.3.x which contains a small bug that is fixed in 1.3.7. Please apply the fix shown in https://github.com/zikula/core/commit/f7e337 in order to fix the error.

10.5.2. Orphan removal with many-to-many relations

If you are generating for 1.3.x please do not enable the *orphanRemoval* property for many-to-many relationships. This feature is supported only in newer versions of Doctrine 2, therefore you need to use 1.4.0 if you want to use it.

10.5.3. The workflows are not fetched properly

Due to a problem in Zikula 1.4.0 the workflow data can not be fetched automatically for an entity yet. Because if the postLoad listener in the entity class (method performPostLoadCallback) calls the workflow the Zikula workflow util class does another selection for this. This leads to both selections getting into a competition about the internal object hydrator of Doctrine. The workflow selection "steals" it from the main selection. So after the first item, when it wants to fetch the second one, it is not possible anymore. In 1.3.x this was not a problem because workflows were called using DBUtil bridged by Doctrine 1.

A clean solution would not use any workflow util class, but join the workflow entity directly. But this affects too many parts to do it quickly. Also to make it really beautiful we would need own workflow tables for each entity (like done with categories for example) in order to keep things together.

Before the initWorkflow() method was called at two places: the entity constructor (for newly-created entities) and the postLoad listener (for fetched entities). Note the Doctrine 2 development is working on a new event which is triggered after all entities have been fetched.

The current workaround is having the initWorkflow() call moved outside the postLoad listener. But now we have to call it elsewhere. Therefore we need something in the controller actions like the following for bypassing the problem:

```
// single entity
$myEntity->initWorkflow();
 // collection of entities
foreach ($entities as $k => $entity) {
    $entity->initWorkflow();
}
```

Note you must call this for all fetched entities, not for new ones (as the call is still included in the constructor).

This solution approach is a bit ugly, because we require you (the mod dev) calling this explicitly from the using code. If you forget it in a certain method workflows are not initialised properly. Of course this will be improved again as soon as Doctrine 2 offers the new event which happens really *post* fetch.

10.5.4. Patch for category selector

Due to a bug in the category selector in Zikula 1.3.5 and 1.3.6 (solved in 1.3.7) you will run into problems when trying to save an entity with categories. To solve this just merge the fix shown in https://github.com/zikula/core/pull/1561/files and you are done.

11. Glossary

11.1. A-C

Term	Description
Abstract syntax	Technology independent part of a domain-specific language, represented by meta models.
AC-MDSD	Architecture-Centric MDSD: a specialisation of MDSD, often used for the domain "software architecture". Covers a domain beginning from the solution space by utilising reference implementations.
Agile Software Development	A principle for managing the development process with agile technologies.
AOP	Aspect-Oriented Software Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
Architecture-Centric MDSD	See AC-MDSD.
Aspect-Oriented Software Development	Development paradigm increasing maintainability of complex system by separated treatment of cross-cutting concerns with aspects.
BNF	Backus-Naur Form
Business use case	Function for integration of object-specific business logic into an overall system.
Cartridge	Modularised generator component which executes a specific task and can be flexibly combined with other components.
Cascaded MDSD	Enhancement of AC-MDSD for separation of different generators into modular cartridges. In this context a cascading defines a sequential combination of several cartridges.
CCC	See Cross-Cutting Concern.
Cheatsheet	Integrated tutorial with interaction elements for an extended user support.
CMOF	Complete MOF
Concrete syntax	Textual, structural, graphical or hybrid notations of a domain-specific language. Serves for describing applications in the form of models by the user.
Constraint	Rule-like definition in context of a meta class. Serves for validation of models which are based on the corresponding meta model.
Cross-Cutting Concern	A function representing a separatable aspect whose implementation affects multiple program parts. A typical example is the logging of errors.
CSS	Cascading Style Sheets
CVS	Concurrent Version System

11.2. D-F

Term	Description
Doctrine	An ORM layer for PHP, represents the interface between objects in an application and the relational data storage behind it.
Domain	A limited area of knowledge or interest, represents a problem space from the real world in most cases.
Domain analysis	Process within the meta modeling, cares for finding commonalities and differences between different members of a software system family.
Domain architecture	From the area of product line engineering, marks the sum of a domain-specific language and the platform on which the applications realised with it are executed.
Domain Specific Language	A language offering precise means for expression for a domain. Contains an abstract syntax as well as concrete notations and transformations for further processing of created models.
DSL	Domain Specific Language
Eclipse Modeling Framework	Flexible and powerful framework for Ecore-based meta modeling.
Ecore	Meta meta model implemented in the Eclipse Modeling Framework, compatible to the Meta Object Facility.
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
Framework	Collection of libraries and functions providing a well-defined frame for implementations of a programming language.
Function point analysis	Procedure for measuring the functional complexity of a software system.

11.3. G-L

Term	Description
Generative software architecture	Specialisation of a domain architecture for encapsulating technical implementation details in the context of architecture-centric MDSD.
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GMT	Generative Model Transformer
Graphical Editing Framework	Eclipse-based framework for the realisation of concrete notations in the form of graphical editors.
Graphical Modeling Framework	Generative bridge between the Eclipse Modeling Framework and the Graphical Editing Framework. Creates graphical editors based on Ecore-based meta models.
GUI	Graphical User Interface
Hook	Special form of a Zikula application, enriches other components by additional functionality.
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
Interceptor	Special method in PHP which is called automatically on certain events.
JET	Java Emitter Templates
JSF	Java Server Faces
JSP	Java Server Pages

11.4. M-N

Term	Description
M2M	Model-to-Model
M2T	Model-to-Text
MDA	See Model-Driven Architecture
MDD	Model-Driven Development
MDSD	See Model-Driven Software Development
Meta class	Instance of a meta model element in the form of a generated class.
Meta meta model	Defines possible elements and relations for meta models.
Meta model	Abstract syntax of a domain-specific language. Is created based on a meta meta model in the meta modeling process.
Meta modeling	Abstract definition of the core concepts of a domain in the form of meta models. Based on findouts in the domain analysis.
Meta Object Facility	Standard for meta meta models, adopted by the Object Management Group.
Model	A simplifying presentation of structures, functionalities and progress forms. Represents an application in the model-driven software development which is described in an abstract form.
Model-Based	In model-based systems models are the ?base? for an application. They serve for documentation and basis for generating a program skeleton.
Model-Driven	Model-driven systems use domain-specific languages in order to be able to create software completely from models.
Model-Driven Architecture	Standard for the realisation of software systems with a separation of technology independent and dependent parts, defined by the Object Management Group.
Model-Driven Software Development	Pragmatic development paradigm for generation of software from models. See also model-driven.
Module	The module [mo?du?le] is a building block, in general a part of a bigger system. In informatics it stands for a unit which is assembled by multiple elements and can be exchanged at any time.
MOF	See Meta Object Facility
MVC	Model View Controller
MWE	Model Workflow Engine

11.5. O-R

Term	Description
Object Constraint Language	Standard language for the definition of constraints, adapted by the Object Management Group.
Object Management Group	Consortium developing different standards for improving interoperability, beside others MDA, UML, Object Constraint Language and Query / View / Transition.
OCL	See Object Constraint Language
OMG	See Object Management Group
ORM	Object-Relational Mapping
PDE	Plugin Development Environment
PDF	Portable Document Format
PHP	PHP Hypertext Processor
PIM	Platform-Independent Model
PLE	See Product Line Engineering
Pointcuts	Program code parts where aspects can embed the handling of cross-cutting concerns.
Product line	A software system family in the context of product line engineering.
Product Line Engineering	A principle comparing software development with industrial production processes and aiming on according automations.
PSM	Product-Specific Model
Query / View / Transition	Standardised language adapted by the Object Meta Group for model-to-model transformations.
QVT	See Query / View / Transition
RAP	See Rich Ajax Platform
RCP	See Rich Client Platform
Reference implementation	Existing application of a software system family. Used in AC-MDSD for deriving a domain's core concepts from the solution space.
Rich Ajax Platform	Ajax port for SWT elements and other parts of the Rich Client Platform.
Rich Client Platform	Generic application framework based on Eclipse. Allows the implementation of portable Java applications.

11.6. S-Z

Term	Description
SDK	Software Development Kit
Smarty	Template engine used in Zikula for a strict separation of content and presentation.
Software Engineering	Engineer discipline for systematic improvement of the quality of software systems.
Software system family	Collection of several applications whose schematic code parts are based on the same architectural principles.
SPL	Standard PHP Library
Sub domain	Functional or technical partition of a domain which deals with the representation of specific aspects of a software system.
SVN	Subversion
SWT	Standard Widget Toolkit
System use case	See Use case.
Transformation	Modification of models leading to other models again or to source code.
UML	See Unified Modeling Language
Unified Modeling Language	Standardised language adapted by the Object Management Group for describing software systems with models. The UML works model-based.
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
Use case	An interaction offered by a software system.
UUID	Universally Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XML Metadata Interchange	XML-based file format for interoperable storage of models.
XP	Extreme Programming
XSD	XML Schema Definition

List of External Links

http://modulestudio.de/en/tutorial/using-the-generator.html
http://modulestudio.de/en/tutorial/validation-instead-of-certification-of-3rd-party-fram
html
https://github.com/zikula/core/issues/118
http://modulestudio.de/en/tutorial/customise-palette.html
https://github.com/zikula/core/tree/master/tools
http://modulestudio.de/en/tutorial/modeling-the-controllers.html
https://github.com/Guite/MostGenerator/issues/5
http://modulestudio.de/en/tutorial/creating-multiple-elements-quickly.html
http://help.eclipse.org/helios/topic/org.eclipse.gmf.doc/prog-guide/runtime/
Developer%20Guide%20to%20Keyboard%20Accessibility.html
https://github.com/Guite/MostGenerator/issues/254
http://modulestudio.de/en/tutorial/moving-fields-with-drag-n-drop.html
http://modulestudio.de/en/tutorial/basic-usage.html
http://en.wikipedia.org/wiki/COCOMO
http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/index.html
http://modulestudio.de/en/tutorial/multiple-container-elements.html
http://modulestudio.de/en/tutorial/structure-and-customisation-of-generated-applications
html
https://github.com/Guite/MostGenerator/issues/30
http://modulestudio.de/en/tutorial/how-mdsd-reduces-costs-for-long-term-maintenance-of-c
html
http://modulestudio.de/en/tutorial/from-scaffolding-and-uml-to-mdsd-and-dsl.
html
http://modulestudio.de/en/tutorial/the-first-zikula-application-in-10-minutes.
html
https://github.com/zikula/core/commit/f7e3379e7060859aab334be6707fe6e0ab61baf8
http://modulestudio.de/en/tutorial/working-with-multiple-windows.html
http://modulestudio.de/en/product/advantages-of-modulestudio.html
http://modulestudio.de/en/tutorial/describing-the-model.html
http://modulestudio.de/en/product/what-is-modulestudio.html
https://github.com/Guite/MostGenerator/issues/29
https://github.com/zikula/core/pull/1561/files
https://github.com/Guite/MostGenerator/issues/46
http://modulestudio.de/en/tutorial/basic-settings-in-main-editor.html
https://github.com/Guite/MostGenerator/issues/87
http://en.wikipedia.org/wiki/Function_point

https://github.com/l3pp4rd/DoctrineExtensions/issues/936